Opis języka AWK

Tomasz Przechlewski

18 listopada 2000 roku

Poszerzona i poprawiona wersja tekstu: B. Lichoński i T. Przechlewski, AWK – opis języka z przykładami [5].

Spis treści

1. Wprowadzenie 5
1.1. Struktura pliku wejściowego 7
1.2. Skrypty wykonywalne 9

2. Wzorce 9
2.1. `BEGIN/END` 10
2.2. Wyrażenie 10
2.3. Wzorzec regularny 11
2.4. Wzorzec złożony 11
2.5. Wzorzec z przecinkiem 12

3. Wyrażenia regularne 12
3.1. Napisy jako wyrażenia regularne 14

4. Wyrażenia 15
4.1. Stałe 15
4.2. Zmienne 15
4.3. Zmienne wbudowane 16
4.4. Zmienne przechowujące zawartości pól 17
4.5. Operatory arytmetyczne 17
4.6. Operatory napisowe 17
4.7. Operatory porównywania i operatory logiczne 18
4.8. Operatory pasowania do wyrażeń regularnych 19
4.9. Przypisanie 19
4.10. Operator warunkowy `?:` 19

5. Arytmetyczne funkcje wbudowane 20

6. Napisowe funkcje wbudowane 20

7. Funkcje daty i czasu 24

8. Instrukcje sterujące 24
8.1. Instrukcja pusta 27

9. Tablice asocjacyjne 27
9.1. Instrukcja `delete` 28
9.2. Tablice wielowymiarowe 28

10. Funkcje 29

11. Wejście **30**
11.1. Pola 31
11.2. Rekordy 31
11.3. Instrukcja `getline` 34
11.4. Pola o ustalonej długości 36

12. Instrukcje wyjścia – `print`/`printf` **36**
12.1. Instrukcja `printf` 36
12.2. Instrukcja `print` 38
12.3. Drukowanie do plików 38
12.4. Drukowanie w potoku 40
12.5. Funkcja `close` 40
12.6. Funkcja `fflush`* 42
12.7. Funkcja `system` 42

13. Argumenty wywołania programu **42**

14. Uruchamianie AWK **43**

Bibliografia **44**

Skorowidz **44**

1. Wprowadzenie

System UNIX wyposażony jest w wiele narzędzi wspomagających pracę użytkowników. AWK jest jednym ze standardowych narzędzi tego systemu, choć implementacje AWK znaleźć można niemal na każdej platformie systemowej. Nazwa AWK pochodzi od inicjałów jego twórców: Alfreda V. Aho, Petera J. Weinbergera i Briana W. Kernighana.

W jednym zdaniu można powiedzieć, że AWK służy do transformacji danych tekstowych. Istotą działania AWK jest przetwarzanie pliku lub plików wejściowych według zadanego zbioru reguł, generując strumień danych wyjściowych, czy też plików wyjściowych.

Każdy program języka AWK składa się z dowolnej liczby par[1]:

⟨*wzorzec*⟩ `{` ⟨*akcja*⟩ `}`

⟨*Wzorzec*⟩ jest wyrażeniem logicznym, które może być prawdziwe (wówczas wykonywana jest ⟨*akcja*⟩) lub fałszywe (⟨*akcja*⟩ nie jest wykonywana). Akcja jest *zawsze* zawarta pomiędzy parą nawiasów `{` i `}`.

AWK może być wywołany na wiele sposobów. Jeżeli program jest krótki, to najprościej jest umieścić go pomiędzy znakami *pojedynczego* cudzysłowa w linii poleceń, w następujący sposób (w systemach DOS/MS Windows zamiast cudzysłowa pojedynczego należy użyć cudzysłowa maszynowego `"`):

`awk '`⟨*program*⟩`'` ⟨*plik1*⟩ ⟨*plik2*⟩ ⟨*...*⟩

Kiedy program jest długi wygodniejsze jest jego umieszczenie w oddzielnym pliku; w tym wypadku uruchomienie programu wygląda następująco (⟨*program*⟩ oznacza nazwę pliku zawierającego program):

`awk -f` ⟨*program*⟩ ⟨*plik1*⟩ ⟨*plik2*⟩ ⟨*...*⟩

Program w języku AWK może zawierać wiele par ⟨*wzorzec*⟩`{` ⟨*akcja*⟩ `}`. AWK czyta po kolei wiersze z ⟨*pliku1*⟩, ⟨*pliku2*⟩ itd. dla wszystkich plików, których nazwy podano w linii poleceń. Pliki te są modyfikowane według programu z pliku ⟨*program*⟩, tj. dla każdego wiersza z każdego z plików wejściowych obliczane są kolejne wzorce (w kolejności ich występowania w programie) i wykonywane akcje. Przykład 1 (s. 6) pokazuje program wykorzystujący 2 wzorce. Uwaga: Przykładowe programy mogą zawierać konstrukcje w danym momencie jeszcze nie *omówione*. Jeżeli coś jest niezrozumiałe, czytaj dalej, a po lekturze całego tekstu wróć do tego miejsca – wszystko powinno być jasne. Uwaga 2: Przedstawione przykłady programów są gotowe do uruchomienia w systemach Unix/Linux natomiast w systemach DOS/MS Windows wymagają czasami modyfikacji, np. zamiany znaków `'` na `"` czy zastąpienia skryptów shellowych odpowiednimi plikami `.bat`.

Przed przedstawieniem bardziej szczegółowych informacji o języku wymienimy kilka podstawowych reguł składni AWK:

[1] W przykładach programów fragmenty, które oznaczają pewne pojęcia, a nie konkretne konstrukcje języka oznaczono kursywą wewnątrz nawiasów trójkątnych, np.: ⟨*instrukcja*⟩ oznacza każdą instrukcję AWK.

- kolejne pary „⟨*wzorzec*⟩ `{` ⟨*akcja*⟩ `}`” muszą być oddzielone średnikami lub znakami nowego wiersza;
- akcje mogą składać się z wielu poleceń, które muszą być oddzielone średnikami lub znakami nowego wiersza;
- wzorzec lub akcja może zostać pominięty. W wypadku braku wzorca akcja zostaje wykonana *dla każdego* wiersza pliku wejściowego. Jeżeli pominiemy akcję, to AWK zastosuje akcję domyślną, jaką jest wydrukowanie wiersza z pliku wejściowego (czyli `{ print $0 }` w składni AWK).

PRZYKŁAD 1
Poniższy program przepisuje plik wejściowy zastępując kolejne puste wiersze, jednym pustym wierszem (autor: Nelson H. F. Beebe).

```
NF == 0 { nb++ }
NF > 0  { if (nb > 0) print ""; nb = 0; print $0; }
```

□

UWAGI: Ponieważ zarówno ⟨*wzorzec*⟩ jak i ⟨*akcja*⟩ są opcjonalne, to stosowanie następującego stylu programowania:

```
⟨wzorzec⟩
{
  ⟨akcja⟩
}
```

jest błędem, gdyż powyższy zapis jest interpretowany jako dwie pary wzorzec-akcja. Pierwszy wiersz jest interpretowany jako ⟨*wzorzec*⟩ nie posiadający jawnie wyspecyfikowanej ⟨*akcji*⟩, co oznacza wydrukowanie wszystkich wierszy z pliku wejściowego, dla których wartością logiczną ⟨*wzorca*⟩ jest prawda. Trzy następne wiersze są natomiast interpretowane jako ⟨*akcja*⟩ bez jawnie podanego wzorca, co powoduje, że będzie ona wykonywana *dla każdego* kolejnego wiersza z pliku wejściowego. Jeżeli ktoś lubi tego typu styl pisania programów, to musi umieścić otwierający nawias `{` w jednym wierszu z odpowiadającym mu ⟨*wzorcem*⟩.

Jest wiele interpretatorów AWK. W systemach uniksopodobnych są dostarczane razem z systemem. Istnieją też wersje ogólnodostępne, takie jak: gawk – firmowany przez Free Software Foundation czy mawk Michaela Brennana. Różne implementacje AWK *nie są* w 100% kompatybilne ze sobą a ponadto wiele z nich posiada rozszerzenia w stosunku do standardu (za standard przyjmujemy opis z [1]). W niniejszym tekście przedstawiono standard AWK i rozszerzenia interpretatora gawk w wersji 3.*.

Doświadczenia autora wskazują, że komercyjne implementacje AWK *są gorsze* niż gawk. Przykładowo maksymalna liczba pól w rekordzie albo maksymalna długość rekordu w znakach może być śmiesznie mała (np. 99 pól w rekordzie). Z tego względu zachęcam do zainstalowania i posługiwania się gawk-iem także w systemach, w których znajduje się inna implementacja AWK. Zainstalowanie gawk-a ze

źródeł jest w większości systemów uniksowych bardzo proste (naprawdę!). Oczywiście w skład systemu GNU/Linux standardowo wchodzi gawk zatem problem z głowy. Użytkownicy systemów DOS czy Microsoft Windows, których producent oczywiście nie dołącza AWK, muszą samodzielnie skopiować „z sieci” i zaistalować gawk-a (lub mawk-a). Kopiujemy gotowe pliki wykonywalne ponieważ ich samodzielne kompilowanie nie jest prostą rzeczą w systemie DOS/MS Windows.

Uwagi: Funkcje, zmienne wbudowane oraz inne konstrukcje poszerzające możliwości standardowego AWK zostały w tekście wyróżnione za pomocą znaku *.

1.1. Struktura pliku wejściowego

Dla AWK dane wejściowe składają się z *rekordów*, które rozdzielone są separatorami `RS`. Standardowo rekordem jest cały wiersz, czyli separatorem jest znak końca wiersza.

Rekordy podzielone są na *pola*, które rozdzielone są separatorami pól `FS`. Domyślnie separatorami pól są odstępy, tj. znaki spacji i/lub tabulacji. Poniżej zamieszczono zawartość pliku `wina.txt`, którego każdy wiersz zawiera: nazwę wina, symbol kraju-producenta, kolor, smak, cenę w złotych oraz liczbę butelek sprzedanych, na przykład w ostatnim miesiącu:

```
Chardonnay Lyngrove   RPA białe wytrawne 95.00 131
Cabernet Sauvignon Merlot Lyngrove   RPA czerwone wytrawne 95.00 58
Baron De France   Fra białe wytrawne 25.00 289
Anjou Blanc Chenin   Fra białe wytrawne 33.00 392
Anjou D'rose   Fra   różowe półwytrawne 33.00 207
Anjou Rouge Cabernet  Fra czerwone półwytrawne 33.00 266
Bordeaux Graveschateau Saint Galier Fra białe wytrawne 89.00 144
Marquis De Chasse 1996 Fra czerwone wytrawne 55.00 229
Don Kichot Tinto Spa czerwone półwytrawne 25.00 360
Rioja Miralcampo   Spa czerwone   wytrawne 62.00 210
Rioja Miralcampo   Spa białe wytrawne 62.00 179
Sherry Rich Cream   Spa czerwone słodkie 109.00  55
Sole D'italia   Ita czerwone wytrawne  25.00 666
Valpolicella   Ita czerwone wytrawne 45.00 370
Chianti Villa Bellafonte  Ita czerwone wytrawne 68.00 131
Asti Spumante Docg   Ita białe półsłodkie 51.00 207
Moscato Spumante Vsq   Ita białe słodkie 26.00 629
```

Ponieważ nazwa wina składa się z wielu wyrazów, liczba pól w poszczególnych rekordach jest zmienna, np. pierwszy wiersz zawiera 7 pól, drugi 9 pól, trzeci 8 pól, itd.

`RS` i `FS` są zmiennymi, a więc można im nadać wartość. Przykładowo, jeśli zmiennej `FS` nadamy wartość ‘`;`’, to separatorami pól będą znaki ‘`;`’ a nie spacje i tabulatory. Wartością zmiennej `FS` może być dowolne *wyrażenie regularne*.

W akcjach i wzorcach do wartości pól można się odwoływać za pomocą zmiennych postaci `$`⟨*nr-pola*⟩. Tak więc `$1` to pierwsze pole rekordu, `$2` drugie itd. `$0` oznacza cały rekord. Wbudowana zmienna `NF` przechowywuje liczbę pól bieżącego rekordu, stąd `$NF` to zmienna zawierająca zawartość ostatniego pola (w każdym rekordzie).

Przykładowo w pliku `wina.txt` zmienna `$NF` zawiera wielkość sprzedaży każdego gatunku wina. Dla pierwszego z win `$1` zawiera napis `"Chardonnay"` a `$2` napis `"Lyngrove"`. Dla drugiego wiersza wartością `$1` będzie `"Cabernet"` itd. Ponieważ liczba wyrazów w nazwie wina jest nieustalona, wydawać by się mogło, że dotarcie do odpowiednich informacji w każdym wierszu może być skomplikowane, ale tak nie jest, bo pola można także liczyć od końca. Zapis `$(NF-1)` oznacza pole *przedostatnie*, `$(NF-2)` drugie od końca itd. Nawiasy okrągłe są tutaj obowiązkowe, a ich brak zmienia znaczenie takiej konstrukcji, więcej szczegółów jest w punkcie 4.4.

Plik `wina.txt` był jednorodny w tym sensie, że każdy rekord (wiersz) zawierał informację o jednym gatunku wina. Często można mieć do czynienia z plikami, które zawierają różne rekordy, np. poniżej przedstawiono plik `tdf2000.txt` zawierający zestawienie wszystkich podjazdów o nachyleniu większym od 5% na alpejskich etapach wyścigu Tour de France '2000:

```
**** Tour de France *** 1/07/2000 -- 23/07/2000 *** 3630km *****
Nazwa góry/przełęczy    Dystans Początek Szczyt Różnica Długość
----------------------------------------------------------------
Etap 14: Draguignan--Briancon
 Col d'Allos              127.5     1432   2250     818    13.4
 Col de Vars              177.5     1401   2109     708    10.4
 Col d'Izoard             249.5     1345   2361    1016    14.1

Etap 15: Briancon--Courchevel
 Col du Galibier           33.0     2065   2645     580     8.4
 Col de la Madeleine      110.5      456   2000    1544    19.3
 Monte de Courchevel      173.5      602   2004    1402    17.3

Etap 16: Courchevel--Morzine
 Col des Saisies           80.0      668   1650     982    15.1
 Col des Aravis           106.5      973   1498     525     8.2
 Col de la Colombiere     131.0      922   1618     696    11.6
 Col de Chatillon         158.0      478    733     255     4.9
 Col de Joux-Plane        196.5      692   1700    1008    12.0
```

W tym przypadku w pliku znajdują się następujące grupy rekordów: nagłówek (wiersze 1–3), określające etap, zawierające dane o każdej górze (nazwa, dystans od startu w kilometrach, wysokość w metrach n.p.m. podnóża i szczytu, różnica poziomów oraz długość podjazdu w kilometrach) i puste wiersze separujące.

Do plików `wina.txt` oraz `tdf2000.txt` będziemy często wracać w przykładach zamieszczonych w dalszej części tekstu.

1.2. Skrypty wykonywalne

Za pomocą mechanizmu znaków `#!` umieszczonych jako dwa pierwsze znaki w pliku możliwe jest uruchamianie programów AWK-owych, tak jakby były programami wykonywalnymi (w systemach uniksowych, nie w DOS/MS Windows!). Jeżeli przykładowo umieścimy w pliku `pr10` następujący kod:

```
#!/usr/bin/awk -f
NR <= 10
```

to, po nadaniu plikowi prawa do wykonywania (za pomocą `chmod`), możemy uruchamiać program pisząc po prostu: `pr10` ⟨*plik*⟩. (Program drukuje pierwsze 10 wierszy ⟨*pliku*⟩.)

2. Wzorce

Wzorce służą do wyznaczenia tych wierszy tekstu, dla których wykonane mają być odpowiednie akcje. W ogólnym wypadku wzorzec może być kombinacją wyrażeń logicznych i wyrażeń regularnych. Ponieważ wzorce są wyrażeniami logicznymi, dozwolone są operatory logiczne: `&&`, `||`, `!` oraz nawiasy. Istnieją dwa specjalne wzorce o nazwie `BEGIN` i `END`. Oto ogólna specyfikacja wzorców:

Wzorce

`BEGIN{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana *przed otwarciem* pliku wejściowego.

`END{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana po zamknięciu pliku (plików) wejściowego.

⟨*wyrażenie*⟩`{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana za każdym razem gdy wartość ⟨*wyrażenia*⟩ jest równa *prawda*, tj. jest niezerowa (dla wyrażeń numerycznych) lub niepusta (dla napisów).

`/`⟨*wyrażenie-regularne*⟩`/{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana za każdym razem gdy wiersz z pliku wejściowego zawiera ciąg znaków pasujący do ⟨*wyrażenia-regularnego*⟩.

⟨*wzorzec-złożony*⟩`{`⟨*akcja*⟩`}`
wzorzec złożony to kombinacja logiczna dowolnych warunków. Można stosować operatory `&&` (koniunkcja), `||` (alternatywa), `!` (negacja) oraz nawiasy. Por. 2.4.

⟨*wzorzec1*⟩, ⟨*wzorzec2*⟩`{`⟨*akcja*⟩`}`
⟨*akcja*⟩ jest wykonywana dla wszystkich wierszy od wiersza zawierającego ⟨*wzorzec1*⟩ do wiersza zawierającego ⟨*wzorzec2*⟩ (łącznie z tymi wierszami). ⟨*Wzorzec*⟩ oznacza ⟨*wyrażenie*⟩ bądź ⟨*wyrażenie-regularne*⟩.

Wzorce `BEGIN` i `END` nie mogą być częścią wzorca złożonego. Podobnie częścią wzorca złożonego nie może być wzorzec z przecinkiem.

2.1. BEGIN/END

Wzorzec `BEGIN` nie pasuje do żadnego wiersza z pliku wejściowego, a odpowiadająca mu akcja jest wykonywana przed przeczytaniem przez AWK pierwszego znaku z tego pliku. Podobnie instrukcje wzorca `END` wykonywane są po przeczytaniu wszystkich znaków pliku wejściowego. Możliwe jest umieszczenie wielu wzorców `BEGIN` i `END` w programie AWK-owym; są one wtedy wykonywane po kolei. Zwyczajowo wzorce `BEGIN` są umieszczane na początku a `END` na końcu pliku.

Jednym z najczęstszych sposobów użycia `BEGIN` jest zmiana domyślnego sposobu w jaki AWK dzieli wiersze z czytanego pliku na pola. Wbudowana zmienna `FS` definiuje napis-separator pól w rekordzie. Domyślnie pola oddzielone są znakami spacji lub/i tabulacji (`FS=" "`). Przy uruchomieniu programu zawierającego tylko wzorce `BEGIN` AWK nie oczekuje w linii poleceń nazwy żadnego pliku wejściowego, por. przykład 17 (s. 35).

2.2. Wyrażenie

Wzorcami mogą być wyrażenia arytmetyczne lub napisowe. Odpowiednia ⟨*akcja*⟩ jest wykonywana za każdym razem gdy takie ⟨*wyrażenie*⟩ ma wartość różną od zera lub od napisu pustego. Przykładowo:

```
$(NF-4) == "Fra" { print $0 } # Wydrukuj wina francuskie
```

w powyższym wierszu, który jest kompletnym programem AWK-owym, wyrażeniem we wzorcu jest porównanie czwartego pola od końca wiersza z napisem `"Fra"`. Jeżeli napisy są identyczne to wartością wyrażenia jest prawda i AWK wykonuje akcję – drukuje cały wiersz. Zgodnie z tym co już powiedziano, tego typu akcja jest wykonywana domyśnie – jeżeli nie podano innej, zatem program można zapisać jeszcze krócej:

```
$(NF-4) == "Fra" # Wydrukuj wina francuskie
```

Przykładem wzorca zawierającego wyrażenie arytmetyczne może być wydrukowanie tych gatunków win, na których obrót był większy od 10 tys. zł:

```
$NF * $(NF-1) > 10000 # Wydrukuj najlepiej kupowane
```

UWAGI: Znak `#` rozpoczyna komentarz – wszystko od znaku `#` do końca wiersza jest przez AWK ignorowane.

2.3. Wzorzec regularny

Wzorzec regularny to wyrażenie regularne ujęte w parę znaków `/`. Podstawowe sposoby użycia wzorca regularnego to:

`/`⟨*r*⟩`/`
Pasuje do bieżącego wiersza z pliku wejściowego jeżeli zawiera ona podnapis pasujący do wyrażenia regularnego ⟨*r*⟩.

⟨*wyrażenie*⟩ `~ /`⟨*r*⟩`/`
Pasuje do napisu będącego wartością ⟨*wyrażenia*⟩ jeżeli zawiera on podnapis pasujący do wyrażenia regularnego ⟨*r*⟩. Zapis `/`⟨*r*⟩`/` jest równoważny formie `$0 ~ /`⟨*r*⟩`/`.

⟨*wyrażenie*⟩ `!~ /`⟨*r*⟩`/`
Pasuje do napisu będącego wartością ⟨*wyrażenia*⟩ jeżeli *nie zawiera* on podnapisu pasującego do wyrażenia regularnego ⟨*r*⟩.

Wyrażenia regularne mogą być pomocne w rozwiązaniu problemu wydrukowania win wytrawnych i półwytrawnych:

```
$(NF-2) ~ /wytrawne/ # wydrukuj coś wytrawnego
```

AWK wydrukuje każdy wiersz, którego trzecie od końca pole zawiera napis `wytrawne`. W tym przykładzie zysk jest niewielki, zamiast `$(NF-2) ~/wytrawne/` można zapisać:

```
$(NF-2) == "wytrawne" # wydrukuj wytrawne
$(NF-2) == "półwytrawne" # ... i półwytrawne
```

ale w ogólnym przypadku wyrażenia regularne potrafią znakomicie ułatwić pracę.

2.4. Wzorzec złożony

Wzorzec złożony to wyrażenie złożone z wzorców i operatorów logicznych `||`, `&&`, `!`. Wzorzec złożony pasuje do bieżącego wiersza z pliku wejściowego jeżeli wartością wyrażenia jest *prawda* (czyli jest niezerowa lub niepusta). Poniższy przykład:

```
$(NF-2) == "wytrawne" || $(NF-2) == "półwytrawne" # coś wytrawnego
```

pokazuje wzorzec złożony i jednocześnie potwierdza, że najlepiej do rozwiązania problemu drukowania win wytrawnych korzystać z wyrażeń regularnych.

Inny przykład wzorca złożonego pozwoli nam rozwiązać problem wydrukowania win francuskich, na których obrót był większy od 10 tys. zł:

```
$(NF-4) == "Fra" && $(NF-1) * $NF > 10000
```

2.5. Wzorzec z przecinkiem

Pasuje do wszystkich wierszy, od wiersza pasującego do ⟨*wzorca1*⟩ do wiersza pasującego do ⟨*wzorca2*⟩ (łącznie z tymi wierszami). Jeżeli w pliku po raz kolejny pojawi się ⟨*wzorzec1*⟩, to znowu pasują wszystkie wiersze aż do napotkania ⟨*wzorca2*⟩. Jeżeli AWK nie znajdzie ⟨*wzorca2*⟩, to pasują wszystkie wiersze aż do końca pliku. Jeżeli w pliku nie ma ⟨*wzorca1*⟩, to nie pasuje żaden wiersz z tego pliku. Przykładowo wykonanie poniższego programu spowoduje wydrukowanie wierszy o numerach od 4 do 14 z pliku `tdf2000.txt`:

```
$3 ~ /Briancon/, $3 ~ /Morzine/
```

Jeżeli wykonamy następujący program:

```
$3 ~ /Draguignan/, $3 ~ /Briancon/
```

to powstaje pytanie czy na wydruku otrzymamy tylko *jeden* wiersz (zawiera `"Draguignan"` i jednocześnie zawiera `"Briancon"`) czy też *sześć* wierszy (od 4 do 9; dziewiąty też zawiera słowo `"Briancon"`) z pliku `tdf2000.txt`? Otóż AWK po sprawdzeniu, że wiersz pasuje do ⟨*wzorca1*⟩ sprawdza *ten sam* wiersz, czy aby nie pasuje on do ⟨*wzorca2*⟩, co powoduje, że w takim wypadku drukowany jest tylko ten wiersz. Gdyby AWK działał tak jak program `sed`, tj. po dopasowaniu wiersza do ⟨*wzorca1*⟩, dopasowywał ⟨*wzorzec2*⟩ do następnego i kolejnych wierszy wtedy na wydruku pojawiłoby się 6 wierszy.

3. Wyrażenia regularne

Wyrażenia regularne to wyrażenia umożliwiające specyfikowanie *klas* napisów. O napisie należącym do tej klasy mówimy, że *pasuje* do wyrażenia regularnego. Wyrażenia regularne są konstruowane z następujących elementów: „normalnych znaków” (wszystkie litery, cyfry, większość pozostałych znaków) oraz metaznaków `\`, `^`, `$`, `.`, `[`, `]`, `|`, `(`, `)`, `*`, `+`, `?`. Poniższa tabela przedstawia poszczególne elementy wyrażeń regularnych, według malejącej kolejności wykonywania:

Wyrażenia regularne

Wyrażenie	Znaczenie
`(`⟨*r*⟩`)`	⟨*r*⟩ (nawiasy służą do grupowania wyrażeń)
`c`	znak nie będący metaznakiem
`\c`	znak sterujący albo znak/metaznak `c`
`^`	początek napisu
`$`	koniec napisu
`.`	dowolny znak
`[ab...]`	dowolny ze znaków `a`, `b`...
`[^ab...]`	dowolny ze znaków oprócz `a`, `b`...
⟨*r*⟩`*`	zero lub więcej powtórzeń ⟨*r*⟩
⟨*r*⟩`+`	jedno lub więcej powtórzenie ⟨*r*⟩

<table>
<tr><td>⟨r⟩?</td><td>zero lub jedno powtórzenie ⟨r⟩</td></tr>
<tr><td>⟨r1⟩|⟨r2⟩</td><td>⟨r1⟩ lub ⟨r2⟩ (⟨r⟩ oznacza wyrażenie regularne)</td></tr>
</table>

Do grupy znaków wewnątrz nawiasów klamrowych pasuje jeden dowolny znak z tej grupy. Wewnątrz nawiasów klamrowych *wszystkie* znaki oprócz `\`, `-` i `^` tracą swoje metaznaczenie. Przykładowo: `[...]` oznacza trzy kropki a nie trzy dowolne znaki.

Zapis `[a-z]` oznacza zakres czyli jeden znak od `a` do `z`, przy czym obowiązuje kolejności kodów ASCII. Zatem specyfikacja `[0-9]` jest równoważna `[0123456789]`, zaś `[A-Da-d]` oznacza `[ABCDabcd]`. Jeżeli znak `-` jest pierwszym znakiem w grupie, wtedy jest traktowany literalnie, tj. `[-+]` oznacza albo minus albo plus podczas gdy `[+-]` jest błędem – AWK oczekuje końca zakresu znaków.

Dopełnieniem grupy lub zakresu znaków jest grupa lub zakres poprzedzona znakiem `^` (bezpośrednio po otwierającym nawiasie `[`). Przykładowo specyfikacja `[^0-9]` oznacza jeden dowolny znak *ale nie* cyfrę; `[^A-ZĄĆĘŁŃÓŚŹŻ]` dowolny znak nie będący dużą literą. Znak `^` jest traktowany literalnie jeżeli nie rozpoczyna grupy. Na przykład `^[^^]` pasuje do każdego znaku *oprócz* znaku `^` na początku napisu.

Nawiasy okrągłe służą do grupowania i – podobnie jak w wyrażeniach arytmetycznych – posiadają najwyższy priorytet wykonania. Przykładowo:

```
/(Ali|ali)(baba|gator)/
```

pasuje do następujących napisów: `"Alibaba"`, `"Aligator"`, `"alibaba"`, oraz `"aligator"`. Zwróćmy uwagę, że ponieważ operator `|` ma najniższy priorytet wykonywania możemy pisać `(Ali|ali)` a nie `((Ali)|(ali))`.

Znaki sterujące, zapisujemy w konwencji języka C. Są to: `\a` (dzwonek, *alarm*), `\b` (znak cofnięcia, *backspace*), `\f` (znak końca strony, *form feed*), `\n` (przejście do nowego wiersza, *new line*), `\r` (*carriage return*), `\t` (znak tabulacji). Ponadto znak `\\` oznacza `\`, zaś każdy znak możemy zapisać przy pomocy kodu ósemkowego używając konwencji `\`⟨*cyfra*⟩⟨*cyfra*⟩⟨*cyfra*⟩.

PRZYKŁAD 2
Do wyrażenia regularnego `/^[ \t]*$/` pasują wszystkie napisy składające się tylko ze znaków spacji, tabulacji i napisu pustego. Do `/^[^ \t]*$/` pasują wszystkie napisy oprócz składających się ze spacji, znaków tabulacji i pustych. Z kolei do `/[+-]?[0-9]+[.]?[0-9]*/` pasują wszystkie liczby rzeczywiste ze znakiem. □

AWK *zawsze* dopasowuje do wyrażenia regularnego najdłuższy z możliwych napisów, rozpoczynając dopasowywanie od lewej strony, tak szybko jak to jest możliwe. Poszczególne operatory powtórzeń (`+`, `*`, `?`) dopasowują napis tak długo jak to jest możliwe. Przykładowo niech plik zawiera następujący krótki tekst:

```
<tr align="left"><td>Chardonnay Lyngrove</td><td>RPA</td></tr>
```

Jaki napis zostanie dopasowany do wyrażenia regularnego `/<.+>/`? Na pierwszy rzut oka mogłoby się wydawać, że `<tr align="left">`, ale nie jest to prawdą:

dopasowany zostanie cały wiersz, gdyż jest to najdłuższy z możliwych pasujących napisów zaczynających się od `<` a kończących `>`.

Rozpatrzmy kolejny przykład. Jaki napis zostanie dopasowany do wyrażenia: `/C*/`? Odpowiedź, że napis rozpoczynający się od `C` w słowie `Chardonnay` do końca wiersza jest błędna. AWK dopasuje się jedynie do napisu o zerowej długości na początku wiersza. Tak się stanie ponieważ 0 powtórzeń `C` jest poprawnym punktem startu a kolejny znak nie jest już dużą literą `C` – dopasowywanie jest kończone z wynikiem 0 `C`.

PRZYKŁAD 3

Rozważmy z kolei jak można z pliku `tdf2000.txt` wydrukować szczyty o różnicy poziomów większej od 1 km. Nie można po prostu napisać `$(NF-1) > 1000` ponieważ nie wszystkie rekordy zawierają w przedostatnim polu różnicę poziomów. Na przykład w pierwszym wierszu przez zupełny przypadek `$(NF-1)` jest równe 3630. Zresztą napotkanie pustego wiersza spowoduje błąd fatalny i przerwanie wykonywania programu ponieważ w AWK *nie wolno* odwoływać się do ujemnych numerów pól. Oczywistym rozwiązaniem jest wyspecyfikowanie akcji dla każdego rodzaju rekordu: nagłówka, nagłówka etapu, góry i pustego:

```
NR==1, /^-+[ \t]*$/ { next }  # Pomiń nagłówek,
NF < 1 { next }               # puste wiersze
/^Etap[ \t]+[0-9]+:/ { next } # oraz nagłówki etapów
$(NF-1) > 1000 # Wydrukuj podjazd o różnicy > 1000
```

Wykorzystana w powyższym programie, jeszcze nie omawiana, instrukcja `next` (por. punkt 8) powoduje: przerwanie wykonywania programu, wczytaniu następnego rekordu ze strumienia danych wejściowych a następnie rozpoczęcie wykonywania programu od początku, tj. od pierwszej pary wzorzec-akcja.

Kluczowe znaczenie w powyższym programie ma kolejność poszczególnych par wzorzec-akcja, dzięki której wzorzec `$(NF-1) > 1000` „widzi” tylko wiersze z danymi o górach, pozostałe zaś są po drodze „odcedzane”. □

3.1. Napisy jako wyrażenia regularne

Zwykle wyrażenia regularne zapisywane są jako ciągi znaków umieszczone pomiędzy znakami ciachów. Możliwe jest też używanie napisów jako wyrażeń regularnych. Wszędzie tam gdzie AWK oczekuje pojawienia się wyrażenia regularnego (jak, np. po prawej stronie operatorów `~` i `!~`) umieszczone tam wyrażenie zostanie przekształcone a następnie zamienione na napis, który będzie interpretowany jako wyrażenie regularne. Przykładowo, poniższy program:

```
BEGIN {cyfry="^[0-9]+$" }; $0 ~cyfry
```

wydrukuje wszystkie wiersze, które zawierają wyłącznie liczbę całkowitą (bez znaku).

Ponieważ wyrażenia napisowe mogą być łączone (por. punkt 4.6), wyrażenie regularne może być konstruowane dynamicznie z części składowych. Przykładowo poniższy program:

```
BEGIN {znak ="[-+]?"; cyfra="[0-9]+"; liczba = "^" znak cyfra "$"}
$0 ~ liczba
```

może służyć do wydrukowane wszystkich wierszy z pliku, które zawierają *wyłącznie* liczbę całkowitą ze znakiem:

UWAGI: Jeżeli wewnątrz wyrażenia regularnego występuje literalnie jakiś metaznak to należy go poprzedzić *dwoma* a nie jednym znakiem `\`. Przykładowo, program:

```
$0 ~ /^\\*[0-9]+\\*$/
```

wydrukuje wiersze zawierające liczbę całkowitą bez znaku otoczoną znakiem `*`.

4. Wyrażenia

Podstawą składni wyrażeń AWK jest składnia wyrażeń języka C wzbogacona o operacje tekstowe. Elementami wyrażeń są: stałe, zmienne, operatory, funkcje wbudowane i definiowane przez użytkownika oraz elementy tablic asocjacyjnych.

4.1. Stałe

W AWK istnieją tylko dwa typy danych: *liczbowy* i *napisowy*. Stałe liczbowe zapisujemy jak w C, tj. `3.1415` lub `1.333e-5`, stałe napisowe otacza się znakami `"`. Stałe napisowe mogą zawierać znaki sterujące takie jak `\n` czy `\f`.

4.2. Zmienne

W składni AWK wyróżniamy zmienne: wbudowane, zdefiniowane przez użytkownika i przechowujące zawartości pól. Nazwy zmiennych definiowanych przez użytkownika mogą składać się z liter, cyfr i znaku podkreślenia. Pierwszym znakiem nazwy *nie może być* cyfra. Nazwy zmiennych wbudowanych składają się wyłącznie z dużych liter alfabetu. Nazwy zmiennych przechowujących zawartości pól zaczynają się od znaku `$`, po którym występuje liczba (ogólnie: wyrażenie).

Zmienne liczbowe przechowują wartości zmiennopozycyjne, przy czym ich dokładność zależna jest od implementacji. Zmienne napisowe przechowują ciągi znaków (napisy). Zmiennych nie deklaruje się. Typ zmiennej określony jest przez kontekst; w razie potrzeby zawsze dokonywana jest odpowiednia konwersja. Zmienna nie zainicjowana ma wartość zero lub `""` (napis pusty).

Interpretacja wyrażeń numerycznych i tekstowych w operacjach logicznych jest następująca: *fałsz* odpowiada liczbie 0 i napisowi pustemu `""`, zaś *prawda* odpowiada wszystkim innym liczbom i napisom.

4.3. Zmienne wbudowane

Zmienne wbudowane są dokładnie opisane przy okazji omawiania tych aspektów AWK, których dotyczą:

Zmienne wbudowane	
Zmienna	Opis znaczenia
`ARGC`	liczba argumentów wywołania programu
`ARGV`	tablica argumentów wywołania programu
`ARGIND`*	indeks w `ARGV` odpowiadający bieżącemu plikowi
`ENVIRON`*	tablica zmiennych środowiskowych
`ERRNO`*	napis z systemowym opisem błędu
`FIELDWIDTHS`*	specyfikacja długości pól, por. punkt 11.4
`FILENAME`	nazwa bieżącego pliku wejściowego
`FNR`	numer bieżącego rekordu w bieżącym pliku
`FS`	separator pól
`IGNORECASE`*	przełącznik rozróżniania wysokości liter
`NF`	liczba pól w bieżącym rekordzie
`NR`	liczba przeczytanych rekordów
`OFMT`	format wydruku argumentów numerycznych funkcji `print`
`OFS`	separator pól na wyjściu, por. punkt 12.2
`ORS`	separator rekordów na wyjściu, por. punkt 12.2
`RLENGTH`	por. opis funkcji `match` w punkcie 6
`RS`	separator rekordów
`RT`*	napis pasujący do wyrażenia `RS`, por. punkt 11.2
`RSTART`	por. opis funkcji `match` w punkcie 6
`SUBSEP`	separator indeksów tablic, por. punkt 9.2

`ENVIRON`* jest tablicą zawierającą wartości zmiennych środowiskowych przy czym indeksami są nazwy zmiennych. Przykładowo:

```
gawk 'BEGIN {print ENVIRON["HOME"] }'
```

zawiera np. `/home/tomek`.

`ERRNO`* zawiera napis z systemowym komunikatem o błędzie, jeżeli przy wykonaniu funkcji `getline` lub `close` wystąpi błąd.

`IGNORECASE`* określa czy AWK rozróżnia duże i małe litery przy porównywaniu napisów i wyrażeń regularnych. Jeżeli `IGNORECASE`* jest niezerowe lub niepuste, wtedy operatory `~`, `!~` i funkcje `gensub`*, `gsub`, `index`, `match`, `split` oraz `sub` nie rozróżniają dużych i małych liter. Dotyczy to także wartości zmiennych `RS` i `FS`. Począwszy od wersji 3.0, gawk obsługuje normę ISO-8859-1 (Latin-1). Standard ten *nie zawiera* jednak większości polskich znaków diakrytycznych.

`ARGIND`* przechowuje indeks, pod którym w tablicy `ARGV` znajduje się nazwa przeglądanego pliku. Zawsze jest prawdziwa równość `FILENAME == ARGV[ARGIND]`.

UWAGI: Zmienna `FILENAME` zawiera nazwę bieżącego pliku wejściowego. Oznacza to, że w obrębie wzorców `BEGIN` i `END` wartość `FILENAME` jest nieokreślona.

4.4. Zmienne przechowujące zawartości pól

Zmienne przechowujące zawartości pól mogą być wykorzystane wewnątrz wyrażeń, można także im nadawać wartości. Jeżeli wartość zmiennej `$0` została zmodyfikowana przez podstawienie lub zamianę (np. funkcją `sub`) to wartości zmiennych `$1`, `$2`, ... oraz zmiennej `NF` są powtórnie wyznaczane. Podobnie jeżeli zmodyfikowano którąkolwiek ze zmiennych `$1`, `$2`, ..., wtedy wartość zmiennej `$0` jest odtwarzana (przy wykorzystaniu wartości zmiennej `OFS` jako separatora pól).

Numery pól mogą być wyznaczane jako wartości wyrażeń. Przykładowo `$(NF-1)` oznacza przedostatnie pole w bieżącym rekordzie (nawiasy są istotne, ponieważ `$NF-1` oznacza wartość pola `$NF` pomniejszoną o 1). Możliwe jest także przypisanie wartości zmiennym odpowiadającym polom nie istniejącym w czytanym pliku. W takim wypadku pole zostanie utworzone i przypisana zostanie odpowiednia wartość. Odwołanie się do pola o numerze ujemnym jest błędem kończącym działanie programu.

4.5. Operatory arytmetyczne

Poniższa tabela zawiera zestawienie wszystkich operatorów arytmetycznych.

Operatory arytmetyczne

Operator	Opis znaczenia
`*`	iloczyn
`+`	suma
`-`	różnica
`/`	iloraz
`%`	modulo
`^`	potęga
`++`	inkrementacja
`--`	dekrementacja

Operator modulo to reszta z dzielenia całkowitego, tj. `5.1 % 3 == 2.1` ma wartość prawda.

4.6. Operatory napisowe

Napisy i zmienne napisowe można łączyć (konkatenować) przy pomocy „niewidocznego” operatora – po prostu należy umieścić napisy obok siebie. Przykładowo po wykonaniu:

```
y = "Ali"; z = "gator"; x1 = y "ba" "ba"; x2=  y z;
```

zmienna `x1` ma wartość „`Alibaba`”; zmienna `x2` ma wartość „`Aligator`”. Oprócz operacji konkatenacji AWK nie ma żadnych innych operatorów napisowych.

4.7. Operatory porównywania i operatory logiczne

Zapis i działanie operatorów w AWK w wypadku zmiennych typu liczbowego jest identyczny jak w języku C. Nowością AWK jest to, że mogą być także stosowane do napisów.

Operatory porównywania

Operator	Opis znaczenia
`==`	równe
`!=`	różne
`<`	mniejsze
`<=`	mniejsze lub równe
`>`	większe
`>=`	większe lub równe

Napisy są porównywane według kodów ASCII w taki sposób, że najpierw porównywane są pierwsze znaki, potem drugie itd. Przykładowo: `"10"` jest mniejsze od `"9"`. Jeżeli jeden napis jest przedrostkiem drugiego to krótszy napis jest mniejszy od dłuższego, np. `"Ali"` jest mniejsze od `"Alibaba"`.

Operatory logiczne

Operator	Opis znaczenia
`&&`	iloczyn
`||`	suma
`!`	zaprzeczenie

PRZYKŁAD 4
Zaimplementujmy funkcję, zwracającą 1 jeżeli rok jest przestępny, lub 0 dla lat nieprzestępnych. Algorytm cytujemy za [4], s. 121.

```
function leapyear(year) {
   return year %4 == 0 && year % 100 \
   != 0 || year%400 ==0; }

BEGIN {print leapyear(1996), leapyear(1806),
   leapyear(1066)}
```

Wzorzec `BEGIN` jest potrzebny tylko dla testowania funkcji. Jeżeli powyższy kod umieścimy, np. w pliku `lyear.awk` to pisząc `awk -f lyear.awk` otrzymamy na ekranie: `1 0 0`. □

UWAGI: Bardzo długa instrukcja może zostać podzielona i zapisana w kilku wierszach. Znakiem kontynuacji jest \, bezpośrednio przed znakiem końca wiersza (por. drugi wiersz przykładu). Jeżeli wiersz kończy się przecinkiem (por. wiersz przedostatni) to znak kontynuacji jest opcjonalny.

4.8. Operatory pasowania do wyrażeń regularnych

W składni AWK są dwa takie operatory `~` oraz `!~`. Umożliwiają one dopasowanie zmiennej do wyrażenia regularnego. Przykładowo `$1 ~/Chardonnay/` jest prawdziwe, gdy pierwsze pole zawiera napis `Chardonnay`. Samotnie pojawiające się wyrażenie `/Chardonnay/` jest równoważne `$0 ~ /Chardonnay/`. Operator `!~` pasuje do dopełnienia wyrażenia regularnego, tj. `$1 !~/Chardonnay/` jest prawdziwe gdy `$1` *nie* zawiera napisu `Chardonnay`.

4.9. Przypisanie

Przypisanie oznaczane jest w AWK pojedynczym znakiem równości `=`. Podobnie jak w języku C operator ten nadaje zmiennej wartość i zwraca przypisaną wartość, stąd dozwolone są wyrażenia postaci `x = y = 1` lub `(x = y) <= 1`.

Z operatorem przypisania związane są operatory modyfikacji: `+=`, `-=`, `*=`, `/=`, `%=`, `/=` i `^=`. Przykładowo wyrażenie `x += y` jest tożsame z `x = x + y`, wyrażenie `x -= y` jest tożsame z `x = x - y` itd.

PRZYKŁAD 5

Dla danych z pliku `wina.txt`, obliczyć obrót dla win białych i czerwonych oraz obrót łączny. Zadanie to rozwiązuje następujący program:

```
#!/usr/bin/awk -f
{obrot = $NF * $(NF-1); oo += obrot }
$(NF - 3) ~ /białe/ {biale += obrot }
$(NF - 3) ~ /czerwone/ {czerwone += obrot }
END { print "Obrót razem:", oo ", w tym: czerwone:",
      czerwone ", białe:", biale "."; }
```

Zwróćmy uwagę na instrukcję `print`. Część argumentów jest oddzielona przecinkiem a część nie. Jest to dopuszczalne ponieważ argumenty oddzielone odstępami są konkatenowane (liczby przed konkatenacją są konwertowane do napisów) i „z punktu widzenia” instrukcji `print` stanowią jeden napis. Natomiast argumenty oddzielone przecinkami są na wydruku oddzielone odstępem. □

4.10. Operator warunkowy ?:

Operator warunkowy `?:` posiada następującą składnię:

⟨*wyrażenie1*⟩ `?` ⟨*wyrażenie2*⟩ `:` ⟨*wyrażenie3*⟩

Najpierw obliczane jest ⟨*wyrażenie1*⟩. Jeśli jest ono prawdziwe obliczane jest ⟨*wyrażenie2*⟩, w przeciwnym wypadku ⟨*wyrażenie3*⟩.

Poniższy program oblicza i drukuje odwrotność pierwszych pól wszystkich rekordów, sprawdzając czy `$1` nie jest równe zeru:

```
{print $1!=0 ? 1/$1 : "Zero w wierszu", NR;}
```

5. Arytmetyczne funkcje wbudowane

AWK posiada inny zestaw funkcji wbudowanych niż język C. Funkcje wbudowane mogą być, bez żadnych ograniczeń, elementami wyrażeń. Oto lista takich funkcji (niech $\langle x \rangle$, $\langle y \rangle$ będą pewnymi wyrażeniami):

Funkcje arytmetyczne

Funkcja	Wartość funkcji
`atan2(`$\langle y \rangle$`,`$\langle x \rangle$`)`	arcus tangens z y/x w zakresia $-\pi$ do π
`cos(`$\langle x \rangle$`)`	cosinus z x, x w radianach
`sin(`$\langle x \rangle$`)`	sinus z x, x w radianach
`exp(`$\langle x \rangle$`)`	eksponent, czyli funkcja wykładnicza e^x
`int(`$\langle x \rangle$`)`	część całkowita z x
`log(`$\langle x \rangle$`)`	logarytm z x przy podstawie e
`sqrt(`$\langle x \rangle$`)`	pierwiastek kwadratowy z x
`rand()`	przypadkowa liczba z przedziału $\langle 0, 1)$
`srand(`$\langle x \rangle$`)`	x wartość początkowa dla generatora liczb pseudolosowych

Używając powyższych funkcji można uzyskać użyteczne liczby, na przykład π lub e: $\text{atan2}(0, -1) = \pi$ oraz $\exp(1) = e$. Również uzyskanie logarytmu dziesiętnego nie jest problemem, jeśli zastosujemy wzór $\log(x)/\log(10)$. Natomiast poprzez podstawienie `randint = int(n * rand()) + 1` nadajemy zmiennej `randint` wartość pseudolosową z przedziału $\langle 1, n \rangle$.

6. Napisowe funkcje wbudowane

Poniższe zestawienie zawiera funkcje AWK umożliwiające manipulowanie napisami. W zestawieniu $\langle r \rangle$ oznacza wyrażenie regularne, $\langle s \rangle$ i $\langle t \rangle$ napis.

Funkcje napisowe

`gsub(`$\langle r \rangle$`,`$\langle s \rangle$`,`$\langle t \rangle$`)`
Zamienia wszystkie napisy pasujące do wyrażenia regularnego $\langle r \rangle$ na napis $\langle s \rangle$ w napisie $\langle t \rangle$. Zwracana jest liczba zamian. Parametry $\langle r \rangle$ i $\langle s \rangle$ mogą być w ogólności wyrażeniami; paramter $\langle t \rangle$ musi być zmienną lub elementem tablicy, tak aby AWK mógł gdzieś przypisać zmodyfikowaną wartość. Jeżeli `gsub` wywołamy tylko z dwoma pierwszymi parametrami, to zmiany dokonywane są w napisie `$0` (tj. `gsub(`$\langle r \rangle$`,`$\langle s \rangle$`)` jest równoważne `gsub(`$\langle r \rangle$`,`$\langle s \rangle$`,$0)`). Znak `&` w $\langle s \rangle$ oznacza *napis, który został dopasowany do* $\langle r \rangle$. Przykładowo po uruchomieniu:

```
awk 'BEGIN {$0 = "baba bababa"; gsub(/(ba)+/,"\\\\&\\&&"); print }'
```

zostanie wydrukowany napis `"\baba&baba \bababa&bababa"`. Ponieważ wewnątrz napisów znak `\` musi być zdublowany żeby AWK traktował go literalnie, dlatego `"\\&"` oznacza `&`. Zaś wstawienie `\` osiągniemy za pomocą `"\\\\"` (cztery `\`!).

`gensub(`$\langle r\rangle$`,`$\langle s\rangle$`,`$\langle a\rangle$`,`$\langle t\rangle$`)`*
Uogólniona funkcja `gsub`. Zwraca zmieniony napis (nie modyfikuje oryginalnego napisu $\langle t\rangle$!). Zamienia napisy pasujące do wyrażenia regularnego $\langle r\rangle$ na $\langle s\rangle$ *w oparciu o* $\langle a\rangle$, w $\langle t\rangle$ (jeżeli nie ma $\langle t\rangle$, domyślnym argumentem jest `$0`). Argumenty $\langle s\rangle$, $\langle a\rangle$ oraz $\langle t\rangle$ są napisami. Argument $\langle a\rangle$ określa, *który* z kolei podnapis pasujący do wyrażenia $\langle r\rangle$ ma być wymieniony. Jeżeli $\langle a\rangle$ jest napisem rozpoczynającym się od `"g"` (lub `"G"`) to wymieniane są *wszystkie* napisy pasujące do $\langle r\rangle$. Funkcja `gensub`* umożliwia wstawienie do $\langle s\rangle$ fragmentów napisu dopasowanego do $\langle r\rangle$. Jeżeli wyrażenie $\langle r\rangle$ podzielimy za pomocą nawiasów, `(` i `)` na części składowe to te składowe mogą później pojawić się w $\langle s\rangle$ (oznaczamy je jako `\`$\langle n\rangle$, gdzie $\langle n\rangle$ jest cyfrą od 1 do 9). Znaczenie tego jest takie, że napis dopasowany do $\langle n\rangle$-tego fragmentu jest kopiowany do napisu zwracanego przez funkcję. W efekcie możliwe są wszelkiego rodzaju *zmiany kontekstowe*, por. przykład 8 (s. 23). Symbol `\0` oznacza napis dopasowany do całego wyrażenia regularnego $\langle r\rangle$ (to samo znaczenie ma `&`). Przykładowo wykonanie programu:

```
awk 'BEGIN {$0 = "12, 122, 1,901, 20,500, 102,153,000";
  $0 = gensub(/([0-9]),([0-9])/, "\\1\\2","g"); print }'
```

spowoduje wydrukowanie napisu `"12, 122, 1901, 20500, 102153000"`. Znaki `&` i `\` są wewnątrz $\langle s\rangle$ specjalne; ich znaczenie podano w opisie funkcji `gsub`. W szczególności należy używać sekwencji `\\` do wstawienia w napisie znaku `\` literalnie.
Poniższy przykład wyjaśnia znaczenie argumentu $\langle a\rangle$:

```
BEGIN{ t = "Alibababa"; print gensub(/ba/, "BA", 2, t) }
```

otrzymamy: `AlibaBAba`

`index(`$\langle s\rangle$`,`$\langle t\rangle$`)`
Zwraca numer pierwszego znaku napisu $\langle t\rangle$ w napisie $\langle s\rangle$. Jeżeli $\langle s\rangle$ nie zawiera $\langle t\rangle$ zwracana jest wartość zero. Pierwszy znak w napisie ma numer 1. Przykładowo: `index("Alibaba", "baba")` zwraca 4;

`length(`$\langle s\rangle$`)`
Podaje długość napisu $\langle s\rangle$.

`match(`$\langle s\rangle$`,`$\langle r\rangle$`)`
Jeżeli $\langle s\rangle$ zawiera podnapis pasujący do $\langle r\rangle$, to zwraca numer pierwszego znaku tego podnapisu; w przeciwnym razie zwracane jest `0`. Ponadto nadawane są wartości zmiennym `RSTART` oraz `RLENGTH`. `RSTART` jest równe wartości zwracanej przez funkcję, `RLENGTH` jest równe długości podnapisu pasującego do $\langle r\rangle$.

`split(`$\langle s\rangle$`,`$\langle a\rangle$`,`$\langle fs\rangle$`)`
Z napisu $\langle s\rangle$ tworzy tablicę napisów $\langle a\rangle$ w oparciu o $\langle fs\rangle$. Argument $\langle fs\rangle$ jest wyrażeniem regularnym. Jeżeli `split` wywołamy tylko z dwoma parametrami to napisem separujący jest wartość zmiennej `FS`, czyli separator pól w rekordzie.

`sprintf(`$\langle format\rangle$`,`$\langle lista\text{-}wyrażeń\rangle$`)`
Zwraca napis, sformatowany według napisu $\langle format\rangle$, por. funkcja `printf`, w punkcie 12.

`sub(`$\langle r\rangle$`,`$\langle s\rangle$`,`$\langle t\rangle$`)`
Funkcja działająca jak `gsub`, ale wymieniająca tylko pierwsze wystąpienie napisu pasującego do $\langle r\rangle$ na napis $\langle s\rangle$. Zwracana jest liczba zamian.

`substr(`$\langle s\rangle$`,`$\langle p\rangle$`,`$\langle n\rangle$`)`
Zwraca napis wycięty z $\langle s\rangle$ począwszy od pozycji $\langle p\rangle$ o długości $\langle n\rangle$ znaków (lub do końca $\langle s\rangle$, jeżeli ostatni argument jest pominięty). Przykładowo wykonanie instrukcji:

```
print substr("Alibaba",4);
```

spowoduje wydrukowanie napisu `"baba"`.

`tolower(`$\langle s\rangle$`)`*
Zwraca napis, w którym duże litery zostały zamienione na małe. W wypadku polskich tekstów funkcja ta ma ograniczone zastosowanie, nie zamieni bowiem liter z górnej połówki tabeli ASCII, gdzie znajdują się `Ą`, `Ć`, `Ę`, itd.

`toupper(`$\langle s\rangle$`)`*
Zwraca napis, w którym małe litery zostały zamienione na duże. Z „polskiego” punktu widzenia ma tę samą wadę co `tolower`*.

PRZYKŁAD 6
Problemem przy wymianie danych pomiędzy DOS-em a Uniksem jest stosowanie innych sekwencji znaków do oznaczania końca wiersza. Poniższy program dokonuje odpowiedniej konwersji:

```
#!/bin/bash
cat $* | awk '{gsub(/\r/,""); print $0}'
```

Wiele uniksowych implementacji AWK źle interpretuje poprawne programy w przypadku gdy przetwarzany plik ma DOS-owe końce wiersza. Na taką okoliczność warto zapamiętać powyższą funkcję `gsub`. □

PRZYKŁAD 7
Poniższa funkcja (porównaj punkt Funkcje dalej w tekście) realizuje kontekstową zamianę frazy na frazę. Jest namiastką tego czego AWK-owi do tej pory brakowało (por. następny przykład) – zamiany wyrażenia regularnego na *wyrażenie regularne.*

```
# wymień w ⟨s⟩ ⟨co⟩ w kontekście ⟨bef⟩ ⟨aft⟩ na ⟨na⟩
function exch (bef, co, aft, na, s) {
   while (match(s, bef co aft) > 0) {
      match(s, bef co aft);
      s = substr(s, 1, RSTART) na substr(s, RSTART+RLENGTH-1); }
   return s;
}
```

Zakładamy, że $\langle bef\rangle$ jest jednym znakiem i poprzedza $\langle co\rangle$. Następujące po $\langle co\rangle$ $\langle aft\rangle$ też jest jednym znakiem. Przykładowo uruchomienie programu:

```
{ s = exch("[0-9]","-","[0-9]", "--", $0);  print s }
```

z podaniem jako argumentu pliku `kloss.txt`:

```
Działalność agenta J-23 (Hans-Peter Kloss) w latach 1941-1945
opisano na stronach 1234-1239.
```

wymieni w nim wszystkie frazy ⟨*cyfra*⟩`-`⟨*cyfra*⟩ na ⟨*cyfra*⟩`--`⟨*cyfra*⟩, pozostałe znaki „`-`” zostaną niezmienione. □

PRZYKŁAD 8
Poniższy program wykorzystujący funkcję `gensub`*:

```
{ $0=gensub(/([0-9])-([0-9])/,"\\1--\\2","g",$0);  print }
```

działa identycznie jak funkcja `exch`, czyli zamienia w całym tekście wszystkie frazy ⟨*cyfra*⟩`-`⟨*cyfra*⟩ na ⟨*cyfra*⟩`--`⟨*cyfra*⟩, np. `1234-1239` zmieni na `1234--1239`. □

PRZYKŁAD 9
Poniższa funkcja jest odpowiednikiem funkcji `toupper`*; ma tę zaletę, że „rozpoznaje” polskie znaki. Łatwo też daje się modyfikować dla różnych wariantów kodowania polskich znaków.

```
function upper(string,     i, j){
   newstring = ""
   for (i = 1; i <= length(string); i++){
      char = substr(string, i, 1)
      for (j = 1; j <= ALPHABET; j++){
         if (char == little[j]){
            char = big[j]; j = ALPHABET + 1; }
      }
      newstring = newstring char;
   }
   return newstring
}

BEGIN{ LOWER = "abcdefghijklmnopqrstuvwxyząćęłńóśźż";
UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZĄĆĘŁŃÓŚŹŻ";
ALPHABET = length(LOWER);

for (i = 1; i <= ALPHABET; i++){
   little[i] = substr(LOWER,i,1)
   big[i] = substr(UPPER,i,1)   } }
```

Dodajmy jeszcze następujący wzorzec `BEGIN`:

```
BEGIN { print upper("Pod źdźbŁEm Żółw spał śnięTY."); }
```

Po uruchomieniu otrzymamy na ekranie: `POD ŹDŹBŁEM ŻÓŁW SPAŁ ŚNIĘTY.` □

7. Funkcje daty i czasu

Interpretator gawk od wersji 3.0 posiada dwie funkcje dotyczące daty i czasu. Są to `systime`* i `strftime`*:

`systime()`

Zwraca bieżący czas w sekundach jakie upłynęły od początku epoki. W standardzie POSIX jest to liczba sekund od 1 Stycznia 1970 r.

`strftime(`⟨*format*⟩`,` ⟨*czas*⟩`)`

Zwraca napis zawierający ⟨*czas*⟩ (liczba w takim samym formacie jak wartość zwracana przez `systime`*) sformatowany według specyfikacji z napisu ⟨*format*⟩. Forma krótka `strftime()`* oznacza użycie formatu `"%a %b %d %H:%M:%S %Z %Y"` oraz bieżącego czasu. Forma `strftime(`⟨*format*⟩`)` wypisuje bieżący czas według specyfikacji z ⟨*formatu*⟩.

Specyfikacje przekształceń funkcji `strftime`* są zgodne ze standardem ANSI C. Każda specyfikacja składa się ze znaku „%" oraz następującego po nim znaku określającego typ konwersji (por. także instrukcja `printf`, w punkcie 12). Nie będziemy podawać pełnej listy znaków konwersji (por. [6], s. 149–151), ograniczymy się do najczęściej stosowanych:

Znaki konwersji

Znak	Typ przekształcenia
`d`	dzień miesiąca (01–31)
`H`	godzina w zapisie 00–23
`I`	godzina w zapisie 01–12
`j`	dzień roku (001–366)
`m`	miesiąc (01-12)
`M`	minuta (00–59)
`S`	sekundy (00–61)
`y`	rok w zapisie dwucyfrowym (00-99)
`Y`	rok w zapisie czterocyfrowym (np. 1066)

Przykładowo w wyniku wykonania poniższego programu:

```
awk 'BEGIN {print "dzisiaj jest:", strftime ("%m:%d:%Y")}'
```

na ekranie zostanie wydrukowany napis `"dzisiaj jest: 08:15:2000"` (ale oczywiście tylko wtedy gdy program uruchomimy 15 Sierpnia 2000 r.)

8. Instrukcje sterujące

AWK pozwala grupować instrukcje, podejmować decyzje (konstrukcja `if-else`) oraz tworzyć pętle (instrukcje `for`, `while`). Składnia tych instrukcji pochodzi bezpośrednio z języka C.

Pojedyncza instrukcja może być zawsze zastąpiona listą instrukcji ujętych w nawiasy grupujące. Na liście instrukcje separowane są znakami końca wiersza lub średnikami. Znaki końca wiersza mogą pojawić się po dowolnym lewym i przed dowolnym prawym nawiasem grupującym.

Spójrzmy przykładowo na składnię instrukcji `if-else`

`if (`⟨*wyrażenie*⟩`)`
⟨*instrukcja1*⟩
`else`
⟨*instrukcja2*⟩

część `else` ⟨*instrukcja2*⟩ jest opcjonalna.

W celu uniknięcia dwuznaczności przyjęto, że każdy `else` jest w parze z bezpośrednio poprzedzającym go `if`-em; przykładowo w następującym fragmencie programu:

```
if (e1) if (e2) s=1; else s=2
```

`else` jest w parze z drugim `if`-em. (Średnik po `s=1` jest wymagany, gdyż `else` znalazł się w jednym wierszu z `if`-em.)

Instrukcje sterujące

`{` ⟨*instrukcje*⟩ `}`
grupowanie instrukcji.

`if (`⟨*w*⟩`)` ⟨*instrukcja*⟩
jeśli wyrażenie ⟨*w*⟩ jest prawdziwe wykonaj ⟨*instrukcję*⟩.

`if (`⟨*w*⟩`)` ⟨*instrukcja1*⟩ `else` ⟨*instrukcja2*⟩
wykonaj ⟨*instrukcję1*⟩ jeśli wyrażenie ⟨*w*⟩ jest prawdziwe, w wypadku przeciwnym ⟨*instrukcję2*⟩.

`while (`⟨*w*⟩`)` ⟨*instrukcja*⟩
jeśli wyrażenie ⟨*w*⟩ jest prawdziwe wykonaj ⟨*instrukcję*⟩ i powtórz.

`for (`⟨*w1*⟩`;` ⟨*w2*⟩`;` ⟨*w3*⟩`)` ⟨*instrukcja*⟩
równoważne instrukcji: ⟨*w1*⟩`; while (`⟨*w2*⟩`) {`⟨*instrukcja*⟩`;` ⟨*w3*⟩`}`.

`for (`⟨*zmienna*⟩ `in` ⟨*tablica*⟩`)` ⟨*instrukcja*⟩
⟨*instrukcja*⟩ jest wykonywana dla ⟨*zmiennej*⟩ przyjmującej kolejno wartości każdego elementu ⟨*tablicy*⟩.

`do` ⟨*instrukcja*⟩ `while (`⟨*w*⟩`)`
wykonaj ⟨*instrukcję*⟩ i jeżeli wyrażenie ⟨*w*⟩ jest prawdziwe powtórz.

`break`
natychmiastowe wyjście z pętli `while`, `for`, `do`.

`continue`
rozpocznij następną iterację w pętlach `while`, `for`, `do`.

`next`
rozpoczęcie następnej iteracji głównej pętli wejściowej. (Przez główną pętlę wejściową rozumiemy mechanizm AWK do analizowania rekord po rekordzie pliku wejściowego.)

`exit` ⟨*wyrażenie*⟩
sterowanie jest przekazywane bezpośrednio do akcji `END`. Jeśli polecenia `exit` użyto w akcji `END`, program kończy działanie. Opcjonalne ⟨*wyrażenie*⟩ zwracane jest jako status zakończenia programu.

`nextfile`*
zakończenie przeglądania bieżącego pliku i przejście do przeglądania następnego z podanych w linii poleceń, lub (dla ostatniego pliku) przejście do wzorca `END`. W wyniku wykonania `nextfile`* zmienia się wartość zmiennej `FILENAME`, wartością `FNR` staje się jeden a wartość `ARGIND`* jest zwiększana o jeden.

PRZYKŁAD 10
Wydrukować z pliku `wina.txt` wino, którego sprzedaż przyniosła największy obrót. Tak postawione zadanie rozwiązuje następujący program:

```
NR == 1 { omax = $NF * $(NF -1); } # inicjalizacja omax
{ o = $NF * $(NF -1);
  if (omax < o ) { omax = o; f = $0; }
  else { if (omax == o) {f = f "\n" $0 } }
}
END {print omax; print f }
```

Ogólny schemat działania programu jest taki, że dla bieżącego wiersza obliczamy obrót i przyrównujemy do dotychczas największego, pamiętanego w zmiennej `omax`. Jeżeli obrót dla bieżącego wiersza jest większy od wartości `omax`, to przypisujemy go jako nową wartość tej zmiennej; zapamiętujemy także wiersz, zawierający tą wartość (`f=$0`).

Drugie polecenie `if` jest potrzebne na wypadek gdyby maksymalny obrót był identyczny dla kilku win. Jest to wprawdzie mało prawdopodobne, ale możliwe. W takim przypadku, kolejne wina są dołączane (konkatenowane) do dotychczasowej wartości zmiennej `f`.

Akcja odpowiadając wzorcowi `NR == 1` jest wykonywana tylko dla pierwszego wiersza i ma na celu inicjalizację zmiennej `omax`. Uważny czytelnik zapewne zauważy, że ponieważ obrót jest zawsze nieujemy, to w tym przypadku nie ma potrzeby jawnej inicjalizacji. Program i tak dawałby poprawne wyniki, bo pierwsze wystąpienie `omax` spowodowałoby przypisanie jej wartości 0. Gdybyśmy jednak chcieli obliczyć zamiast obrotu maksymalnego minimalny, to ten sposób inicjalizacji zmiennej może się przydać. □

8.1. Instrukcja pusta

Jeśli w wierszu programu AWK-owego umieścimy samotny znak '`;`', to otrzymamy instrukcję pustą. Spójrzmy na poniższy program wykorzystujący taką instrukcję w pętli `for`; program drukuje wszystkie wiersze, które zawierają puste pole.

```
BEGIN { FS = "\t" } # Pola oddziela znak tabulacji!
{ for (i=1; i <= NF && $i!=""; i++)
   ;
  if (i <= NF) { print }
}
```

9. Tablice asocjacyjne

Tablice asocjacyjne to jedyny rodzaj tablic dostępny w AWK. Tablic nie trzeba deklarować, określać ich wymiarów czy typu elementów składowych. Utworzenie elementu tablicy następuje w chwili wykonania podstawienia, np. `a[44] = 3.14` lub `a[1]="Alibaba"`, albo innego odwołania do niego. Element nie zainicjowany jest równy zero lub równy napisowi pustemu. *Indeksy nie są liczbami ale napisami.* Użycie w kontekście indeksu liczby spowoduje jej konwersję do odpowiedniego napisu. Trzeba o tym pamiętać; np. `print a["01"]` nie spowoduje wydrukowania słowa `Alibaba` (tylko przypuszczalnie napis pusty) – napisy `"1"` oraz `"01"` są oczywiście różne, podczas gdy liczby nie.

Poniższy program zapamiętuje wszystkie słowa pliku wejściowego oraz liczbę ich wystąpień.

```
{for (i=1; i <= NF; i++) {ls[$i]++}}
```

Zwróćmy uwagę, że za każdym razem, gdy pojawia się nowy wyraz, AWK tworzy nowy element tablicy z wartością początkową 0. Następnie operator inkrementacji nadaje mu wartość 1. Każde następne pojawienie się tego wyrazu powoduje zwiększenie wartości już istniejącego elementu.

Powstaje problem jak dobrać się do elementów tablicy `ls`, skoro nie znamy indeksów (wyrazów tekstu). Do tego celu służy specjalna forma pętli `for`:

`for (`⟨*zmienna*⟩ `in` ⟨*tablica*⟩`)` ⟨*instrukcja*⟩

⟨*zmienna*⟩ przyjmuje iteracyjnie wszystkie wartości indeksów ⟨*tablicy*⟩. Kolejność przeglądania tablicy nie jest ustalona i jest zależna od konkretnej implementacji AWK. Działanie pętli jest *nieokreślone* jeżeli wewnątrz pętli zostaną dodane kolejne elementy do ⟨*tablicy*⟩. Chcąc wydrukować listę słów z omawianego przykładu możemy posłużyć się następującą akcją z wzorcem `END`:

```
END {for (word in ls)  print word, ls[word]  }
```

Wyrażenie:

⟨*indeks*⟩ `in` ⟨*tablica*⟩

pozwala ustalić czy określony ⟨*indeks*⟩ występuje w ⟨*tablicy*⟩. Jeżeli występuje, to wartością wyrażenia jest 1, w wypadku przeciwnym 0. Przykładowo, poniższa instrukcja sprawdza czy w tablicy `ls` wystąpiło słowo `Alibaba`:

```
if ("Alibaba" in ls) {print "OK!"}
else {print "KO!"}
```

9.1. Instrukcja `delete`

Element tablicy możemy usunąć za pomocą instrukcji `delete`:

`delete` ⟨*tablica*⟩`[`⟨*indeks*⟩`]`

przykładowo `delete ls["Alibaba"]` usuwa z tablicy `ls` element odpowiadający indeksowi `"Alibaba"`.

PRZYKŁAD 11

Za pomocą tablic można bardzo sprawnie dokonywać różnego rodzaju obliczeń i zestawień. Poprzednio przekonaliśmy się jak łatwo można policzyć łączny obrót dla win białych albo czerwonych. Równie proste jest policzenie obrotów dla innych kategorii win. Stosowane wtedy rozwiązanie problemu jest jednak kłopotliwe w sytuacji gdy zamiast zestawienia dla jakiegoś smaku czy smaków potrzebne jest zestawienie obrotów z rozbiciem na wszystkie kategorie smakowe. Po pierwsze nie wiemy ile jest kategorii i możemy nie znać ich wartości. Po drugie kategorii może być dużo. Wykorzystując tablice asocjacyjne nie musimy posiadać tych wszystkich informacji, popatrzmy na zadziwiająco krótki program rozwiązujący problem:

```
{ obrot[$(NF-2)] += $NF * $(NF-1) }
END { for (wino in obrot) { print wino, obrot[wino] }}
```

□

9.2. Tablice wielowymiarowe

Tablice wielowymiarowe są symulowane przez AWK za pomocą tablic jednowymiarowych. Z punktu widzenia użytkownika nie ma to wielkiego znaczenia. Przykładowo w wyniku działania poniższego fragmentu programu:

```
for (i=1; i <= 10; i++)
   for (j=1; j <= 10; j++)
      r[i,j] = rand();
```

zostanie utworzona tablica 100 elementów, do których możemy się odwoływać za pomocą par zmiennych indeksowanych postaci `i,j`. Wewnętrznie jednakże poszczególne elementy tablic są indeksowane za pomocą napisów postaci `i SUBSEP j`. Zmienna wbudowana `SUBSEP` przechowuje znak używany do oddzielenia indeksów składowych; standardową wartością tej zmiennej nie jest przecinek ale znak `"\034"`. Sposób testowania czy element `i,j` należy do tablicy nie zmienia się:

```
for ((i,j) in r) {...}
```

zaś w wypadku pętli `for` piszemy:

```
for (k in r) {print r[k]}
```

i, jeżeli jest to potrzebne, wykorzystujemy konstrukcję `split(k,x,SUBSEP)` do dostępu do wartości zmiennych indeksowych.

UWAGI: Elementy tablic nie mogą być tablicami.

PRZYKŁAD 12
Wykorzystując tablice wielowymiarowe nie jest trudną rzeczą wykonanie zestawienia obrotów dla win według dwu kryteriów podziału: smaku i koloru. Poniżej przykładowe rozwiązanie tak postawionego problemu:

```
{  o = $NF * $(NF-1); # obliczamy obrót
   smaki[$(NF-2)] += o; # obrót wg. smaków
   kolory[$(NF-3)] += o; # obrót wg. kolorów
   obrot[$(NF-2), $(NF-3)] += o # obrót wg. smaków i kolorów
}
END { for (smak in smaki) {
         print smak, "...", smaki[smak];
         razem +=smaki[smak];
         for (kolor in kolory) {
              if ((smak, kolor) in obrot)
                print "--", kolor, "...", obrot[smak, kolor];
       } }
    print "Razem ...", razem;
}
```

Na wydruku otrzymamy zestawienie obrotów według smaków i kolorów, łączne obroty według smaków i obroty razem. Program jest króciótki ma jednak poważną wadę: porządek drukowania zestawienia jest dowolny. Poprawieniem tego feleru zajmiemy się w przykładzie 20 (s. 39) po omówieniu polecenia `printf`. □

10. Funkcje

AWK umożliwia definiowanie własnych funkcji. Definicja funkcji może być umieszczona w dowolnym miejscu programu, pomiędzy kolejnymi parami wzorzec-akcja. Funkcje są definiowane następująco:

`function` ⟨*nazwa*⟩ `(`⟨*lista-argumentów*⟩`) {`
   ⟨*lista-instrukcji*⟩ `}`

⟨*lista-argumentów*⟩ to ciąg oddzielonych przecinkami argumentów funkcji. Podczas wywołania funkcji, argumentom nadawane są odpowiednie wartości. Nazwy argumentów są lokalne dla funkcji; są one przekazywane przez wartość, z wyjątkiem

tablic, które są przekazywane „przez referencję”. Argumenty funkcji są *jedynymi* zmiennymi lokalnymi w AWK.

⟨*Lista-instrukcji*⟩ może zawierać instrukcję **return** ⟨*wyrażenie*⟩. Wykonanie **return** polega na obliczeniu wartości ⟨*wyrażenia*⟩, a następnie przekazaniu tej wartości w miejsce wywołania funkcji (tzw. wartość zwracana przez funkcję). ⟨*Wyrażenie*⟩ jest opcjonalne – jeżeli go nie ma, instrukcja **return** jedynie przekazuje sterowanie do miejsca wywołania. Jeżeli wśród ⟨*listy-instrukcji*⟩ nie ma **return** to po wykonaniu ostatniej instrukcji (przed zamykającym nawiasem klamrowym) sterowanie jest przekazywane do miejsca wywołania, a wartość zwracana jest nieokreślona. Zilustrujmy to prostym przykładem funkcji `max` zwracającej większy ze swoich dwu argumentów ([1], s. 53):

```
function max(x, y) { return x > y ? x: y }
```

Funkcje zdefiniowane za pomocą polecenia `function` mogą być użyte w dowolnym wyrażeniu, a także wewnątrz innych funkcji; dozwolona jest także rekursja (por. przykład 13 (s. 30)). Przy wywołaniu funkcji *nie można* umieszczać odstępu pomiędzy jej nazwą a rozpoczynającym listę argumentów nawiasem `(`.

Jak już mówiliśmy *tylko* argumenty funcji są zmiennymi lokalnymi. Wszystkie inne zmienne są *globalne*. Jeżeli chcemy aby AWK „widział” jakąś zmienną tylko lokalnie to jedyną metodą jest jej umieszczenie na liście parametrów przy definiowaniu funkcji. Po prostu nadmiarowe parametry umieszczamy na końcu listy. Nie będą one wykorzystywane do przekazywania wartości lecz będą stanowić dodatkowe zmienne lokalne. Wywołanie funkcji z mniejszą od deklarowanej liczbą parametrów jest w AWK poprawne – wszystkie nadmiarowe parametry przyjmują wartość równą zero lub napisowi pustemu w zależności od kontekstu.

PRZYKŁAD 13
Następująca funkcja rekurencyjna odwraca napis poczynając od znaku `s` ([6], s. 155).

```
function rev(str, s) {
   if (s == 0) return ""
   return (substr(str, s, 1) rev(str, s-1))
}
```

Dla wypróbowania dopiszmy następujący prosty program:

```
BEGIN {print rev("Karuzela",length("Karuzela"))}
```

□

11. Wejście

AWK może czytać dane wejściowe na kilka sposobów. Najprostszym jest uruchomienie go w standardowy sposób czyli, np:

`awk -f` ⟨*program*⟩ ⟨*plik*⟩

W takim kontekście, zgodnie z tym co już napisano wcześniej, AWK czyta ⟨*plik*⟩ wiersz po wierszu. Jeżeli nie podamy ⟨*pliku*⟩ to AWK będzie czekał na strumień danych ze standardowego wejścia (klawiatury). Często jest to działanie niezamierzone – `^C`, `^D`, `^break` czy `^Z` kończą działanie programu w takiej sytuacji.

11.1. Pola

Standardową wartością wbudowanej zmiennej `FS` jest `" "` (spacja – odstęp). W takiej sytuacji poszczególne pola są rozdzielone odstępami lub znakami tabulacji. Sposób rozdzielania pól można zmienić przypisując zmiennej `FS` odpowiedni napis. Jeżeli napis ten jest *dłuższy niż jeden znak*, to AWK traktuje go jako *wyrażenie regularne*. Najdłuższe (*leftmost longest*) ciągi znaków, nie zachodzące na siebie (*non overlapping*), pasujące do tego wyrażenia regularnego będą oddzielać poszczególne pola w bieżącym wierszu. Przykładowo deklaracja:

```
BEGIN {FS ="[,;:]"}
```

powoduje, że pola będą rozdzielane przecinkiem, średnikiem lub dwukropkiem. Kiedy wartością `FS` jest pojedynczy znak (inny od odstępu), to ten znak jest używany do rozdzielania pól.

UWAGI: Wartość zmiennej `FS` może zostać nadana także z poziomu uruchomienia AWK za pomocą przełącznika `-F`. Przykładowo zamiast powyższego wzorca `BEGIN` moglibyśmy napisać w linii poleceń:

`awk -F'[,;:]' -f` ⟨*program*⟩ ⟨*plik*⟩

11.2. Rekordy

Wartość zmiennej `RS` przechowuje napis używany przez AWK do oddzielania poszczególnych rekordów. Standardowo rekordy są oddzielane znakami końca wiersza (co odpowiada przypisaniu `RS="\n"`). W ograniczonym zakresie możemy zmienić sposób w jaki AWK wyróżnia poszczególne rekordy nadając odpowiednią wartość zmiennej `RS`. W opisie [1] separatorem rekordu może być tylko napis jednoznakowy lub napis pusty. Jeżeli `RS=""` (napis pusty), wtedy separatorami rekordów są puste wiersze (jeden lub więcej).

PRZYKŁAD 14

Nie należy do rzadkości sytuacja, w której rekordy zajmują więcej niż jeden wiersz; przykładem może być baza adresowa czy też inny tego typu odpowiednik tradycyjnej kartoteki papierowej. Jeżeli wielkość bazy nie jest zbyt duża, to zamiast specjalizowanych systemów bazodanowych z powodzeniem możemy do jej zarządzania (wyszukiwanie rekordów, drukowanie zestawień itp.) wykorzystywać AWK.

Załóżmy, że plik zawiera informacje adresowe o znajomych osobach: imię i nazwisko, adres i numer telefonu, według schematu, który ilustruje poniższy fragment:

```
Jan Wacław Gdański
ul. Bolesława Krzywoustego 123/3
Gdynia 80-745
+4856 620-75-21

Wanda Kazimiera Matysek
ul. Bohaterów Monte Cassino 2/2
Sopot 81-825
+4858 551-06-50

Wojciech Strzelecki
Aleja Zwycięstwa 5/2
Gdańsk 80-952
+4858 501-26-11
```

Ponieważ poszczególne rekordy są oddzielone pustymi wierszami, wystarczy odpowiednio zmienić separator rekordów, żeby było można manipulować nimi bezpośrednio; przykładowo program:

```
BEGIN {RS=""; ORS="\n\n" }; /Aleja Zwycięstwa/
```

wydrukuje wszystkie rekordy zawierające napis `Aleja Zwycięstwa`. Zwróćmy uwagę na przypisanie `ORS="\n\n"`; bez niego zabrakłoby pustego wiersza pomiędzy kolejnymi drukowanymi rekordami. □

W niektórych implementacjach AWK separator rekordu może być wyrażeniem regularnym. W takiej sytuacji, każdy napis pasujący do tego wyrażenia wyznacza koniec kolejnego rekordu (z tym, że napis ten *nie jest* częścią tego rekordu, podobnie jak w wypadku gdy separatorem jest pojedynczy znak).

Jeżeli `RS` jest wyrażeniem regularnym wtedy zmienna `RT`* przechowuje dla bieżącego rekordu, napis będący jego separatorem od rekordu następnego.

PRZYKŁAD 15
Następujący program ([6], s. 243–244) jest AWK-ową implementacją *edytora potokowego*, tj. takiego programu, który czyta strumień danych, modyfikuje go i wysyła dalej. Sposób użycia jest następujący:

`gawk -f awksed.awk` ⟨*co*⟩ ⟨*naco*⟩ ⟨*plik1*⟩ ⟨*plik2*⟩ ⟨*...*⟩

Spowoduje zastąpienie frazy (wyrażenia regularnego) ⟨*co*⟩ na napis ⟨*naco*⟩ (oba argumenty są wymagalne), w plikach ⟨*plik1*⟩ ⟨*plik2*⟩.... Jeżeli nie podamy listy plików dane będą czytane ze standardowego wejścia.

```
function usage() {
   print "awksed co naco pliki..." > "/dev/stderr"
   exit 1 }

BEGIN {   # sprawdź argumenty wywołania
```

```
   if (ARGC < 3) { usage() }
   RS = ARGV[1]; ORS = ARGV[2]

   # nie używaj argumentów jako nazw plików
   ARGV[1] = ARGV[2] = ""
}
{  if (RT == ""){ printf "%s", $0 }
   else { print } }
```

Idea działania jest prosta: separatorem rekordów jest ⟨*co*⟩ a separatorem rekordów na wyjściu ⟨*naco*⟩. Problemem jest jedynie sytuacja, w której ostatni rekord nie kończy się napisem pasującym do `RS`. Jeżeli plik nie kończy się napisem pasującym do `RS` to zmienna `RT`* będzie równa napisowi pustemu. Stąd, warunek `if (RT=="")...` gwarantuje wydrukowanie całej zawartości pliku wejściowego.

Drugi ciekawy fragment tego przykładu to przypisanie `ARGV[1] = ARGV[2] = ""`. Chodzi o to, żeby AWK nie traktował napisów ⟨*co*⟩ i ⟨*naco*⟩ jako nazw plików wejściowych. Dokładne wyjaśnienie znaczenia tego wiersza znajduje się w punkcie 13. □

PRZYKŁAD 16
Uważny czytelnik zauważył, że przeglądanie bazy z przykładu 14 (s. 31) nie jest wolne od błędów, np. podając:

```
BEGIN {RS=""; ORS="\n\n" }; /Gdańsk/
```

nie tylko zostaną wydrukowane osoby z Gdańska, ale także `Jan Wacław Gdański`. Żeby uniknąć podobnych błędów musimy w jakiś sposób określić znaczenie poszczególnych fragmentów rekordu. Możemy ustalić, np. że imię i nazwisko zajmuje pierwszy wiersz, adres drugi, telefon trzeci itd. Można też zastosować schemat klucz-wartość, według poniższego przykładu:

```
kto Jan Wacław Gdański
tel +4856 620-75-21
email js.gdanski@rugger.gdynia.pl

kto Wanda Kazimiera Matysek
tel +4858 551-06-50

kto Wojciech Strzelecki
miasto Gdańsk 80-952
adres Aleja Zwycięstwa 5/2
tel +4858 501-26-11
email w.strzelecki@ws.com.pl
```

W ten sposób rekordy mogą zawierać różne pola a ich porządek jest dowolny. Pozwala to na łatwe modyfikowanie bazy za pomocą zwykłego edytora tekstowego; nie musimy pamiętać struktury bazy – jest ona samoidentyfikująca się. Łatwo także

dodawać, w miarę potrzeb, nowe pola. A jak przeglądać takie pliki? Jednym ze sposobów jest wykorzystanie tablic asocjacyjnych:

```
#!/bin/bash
cat $3 | awk -vCO=$1 -vCOCO=$2 '
BEGIN {FS="\t" } # Pierwsze pole od reszty oddziela tabulator
NF > 0 {dane[$1] = $2 }
NF < 1 && dane[CO] ~ COCO { drukuj(); usun() }
END { if (dane[CO]~ COCO) {drukuj() }} # ostatni rekord!
function usun(i){ for (i in dane) {delete dane[i] } }
function drukuj(){ print dane["kto"]; print dane["tel"] }'
```

Działanie programu jest następujące: kolejne wiersze zapamiętujemy w tablicy `dane` według schematu: `dane[`⟨*klucz*⟩`] =` ⟨*wartość*⟩. Aby uprościć czytanie plików umawiamy się, że pierwsze pole jest oddzielone od następnych znakiem tabulacji. Po napotkaniu pustego wiersza (`NF<1`) sprawdzamy czy odpowiedni element tablicy pasuje do wzorca poszukiwań (`dane[CO] ~ COCO`). Jeżeli tak, to wykonywana jest procedura drukowania rekordu (funkcja `drukuj`) a następnie usuwamy wszystkie elementy z tablicy `dane` (funkcja `usun`). Po ostatnim rekordzie może nie być pustego wiersza, stąd konieczność dodania wzorca `END`. Zmiennym `CO` (nazwa pola) i `COCO` (wartość pola) przypisujemy wartości wykorzystując opcję `-v` wywołania, por. punkt 14. Dla wygody program został umieszczony w skrypcie shellowym, więc jego uruchomienie wygląda następująco (`szukaj` jest nazwą skryptu):

`szukaj miasto Gdańsk` ⟨*plik-adresowy*⟩

W rezultacie zostanie wydrukowany wyłącznie `Wojciech Strzelecki`. □

11.3. Instrukcja `getline`

Polecenie `getline` umożliwia czytanie danych z bieżącego lub/i z innego pliku tekstowego albo z potoku generowanego przez inny program. Poniżej zestawiono różne formy użycia `getline` (⟨*plik*⟩ i ⟨*program*⟩ to zmienna lub stała napisowa zawierająca odpowiednio nazwę pliku lub programu, z którego AWK ma czytać strumień danych).

`getline`	
Postać instrukcji	Inicjalizowane zmienne
`getline`	`$0`, `NF`, `NR`, `FNR`
`getline` ⟨*z*⟩	⟨*z*⟩, `NR`, `FNR`
`getline <` ⟨*plik*⟩	`$0`, `NF`
`getline` ⟨*z*⟩ `<` ⟨*plik*⟩	⟨*z*⟩
⟨*program*⟩ `| getline`	`$0`, `NF`
⟨*program*⟩ `| getline` ⟨*z*⟩	⟨*z*⟩

Dwie pierwsze formy dotyczą czytania danych z bieżącego pliku, dwie następne z ⟨*pliku*⟩. Dwie ostatnie to wczytywanie danych ze strumienia generowanego

przez inny ⟨*program*⟩. Polecenie `getline` zwraca wartość 1 jeżeli wczytany został następny wiersz tekstu (rekord), 0 jeżeli napotkano koniec pliku (strumienia) danych oraz -1 w wypadku napotkania błędu (np. otwarcia pliku). Jeżeli po słowie `getline` występuje zmienna ⟨*z*⟩ to wczytany wiersz jest dostępny jako wartość tej zmiennej, w innym wypadku jest dostępny jako wartość zmiennej `$0`. Przykładowo pętla:

```
while (getline < ⟨plik⟩ > 0) {...}
```

Jest „tradycyjną” techniką umożliwiającą przejrzenie całego ⟨*pliku*⟩. Poszczególne wiersze dostępne są w każdej iteracji pętli jako wartości zmiennej `$0`.

Podobnie wygląda czytanie danych z potoku. Przykładowo chcąc przekazać do AWK zawartość bieżącego katalogu możemy się posłużyć następującą pętlą:

```
while ("ls -l" | getline > 0) {...}
```

Pliki i potoki otwarte przez AWK są automatycznie zamykane z chwilą zakończenia działania programu. Jeżeli jednak musimy przeglądać wielokrotnie zawartość jakiegoś pliku w obrębie jednego programu AWK-owego to za każdym razem należy zamknąć czytany plik używając funkcji `close`, np.:

```
close (⟨plik⟩); close("ls -lrt")
```

PRZYKŁAD 17
Poniższy program wyświetli listę wszystkich plików większych od `DUZY`.

```
BEGIN { niema = 1; DUZY = 1000000
   while ("ls -l" | getline)
      if (NF == 9  && $5 > DUZY) {print $0; niema =0 }
   if (niema) {print "Nie ma plików większych od", DUZY; }
}
```

Instrukcja `if` najpierw testuje czy wczytany wiersz zawiera dokładnie 9 pól co pozwala „odcedzić” pola nie zawierające informacji o plikach (nagłówek listy). Następnie sprawdzany jest warunek czy wielkość pola jest większa od `DUZY`. □

PRZYKŁAD 18
Poniższy funkcja pobiera bieżącą datę z komputera i udostępnia ją w trzyelementowej tablicy `DAT`, której element `"year"` zawiera rok, element `"mon"` – miesiąc a element `"day"` – dzień.

```
function getdate(x, date) {
   "date" | getline x;  split(x, date, " ");
   DAT["year"]= date[6]; DAT["mon"]= date[2]; DAT["day"]= date[3];
   return;
}
# Przykład wykorzystania funkcji getdate:
BEGIN {getdate(); print DAT["mon"], DAT["day"] }
```

Argumenty x, i date nie służą do przekazywania wartości, ale do uczynienia obu zmiennych lokalnymi (por. uwagi na ten temat w punkcie 10) – funkcję wywołujemy po prostu getdate(). Zwracamy uwagę, że program jest „nieodporny" na zmianę formatu drukowanej daty. □

11.4. Pola o ustalonej długości

Rekord może być także dzielony na pola o ustalonej długości. W tym celu zmiennej wbudowanej FIELDWIDTHS* przypisujemy napis zawierający ciąg oddzielonych odstępami liczb. Każda liczba oznacza długość odpowiedniego pola w znakach. Jeżeli wartością zmiennej FIELDWIDTHS* nie jest napis pusty, pola wyznaczane są w oparciu o specyfikację podaną w tej zmiennej, a nie w oparciu wartość separatora pól (czyli zmienną FS). Przypisanie wartości zmiennej FS (np. FS=FS) przywraca standardowy sposób wyznaczania pól.

PRZYKŁAD 19
Rekordy dotyczące podjazdów w pliku tdf2000.txt mają pola o ustalonej długości, zatem do ich podzielenia można wykorzystać zmienną FIELDWIDTHS*:

```
# Drukuj nazwę góry i jej długość
BEGIN  { FIELDWIDTHS = "1 25 5 5 4 3 4 4 4 4 4" }
NR==1, /^-+[ \t]*$/ { next }
/^Etap[ \t]+[0-9]+:/ { next }
NF > 0 { print $2, $11 }
```

W rezultacie wykonania programu zostaną wydrukowane nazwy gór i długości podjazdów. □

UWAGI: Zwróćmy uwagę, że gawk nie dokonuje żadnego sprawdzenia poprawności specyfikacji podanej w napisie FIELDWIDTHS*.

12. Instrukcje wyjścia – print/printf

Instrukcje print oraz printf służą do drukowania. Pierwsza z nich drukuje swoje argumenty zawsze według tego samego formatu, druga umożliwia precyzyjniejsze sterowanie postacią wypisywanych danych. Dane mogą być drukowane na ekran, do pliku lub w potoku.

12.1. Instrukcja printf

Składnia instrukcji printf jest niemalże identyczna z odpowiednią funkcją języka C. Ogólna postać tej instrukcji jest następująca:

printf (⟨*format*⟩, ⟨*arg1*⟩,⟨*arg2*⟩,...)

Nawiasy (oraz) są opcjonalne. Napis lub zmienna napisowa ⟨*format*⟩ określa sposób przekształcania i formatowania argumentów. Zawiera on dwa rodzaje obiektów: zwykłe znaki, kopiowane po prostu przy drukowaniu oraz specyfikacje przekształceń, z których każda określa sposób przekształcenia i wypisania kolejnego argumentu funkcji `printf`. Specyfikacja ta rozpoczyna się od znaku `%` a kończy znakiem określającym typ konwersji. Pomiędzy nimi możemy użyć ponadto następujących znaków modyfikujących:

- `-`, zawartość pola jest justowana do lewego krańca pola;
- ⟨*ciąg-cyfr*⟩, określający minimalny rozmiar pola (w znakach). Przekształcony argument będzie wpisany do pola o długości *co najmniej* równej ⟨*ciąg-cyfr*⟩. Jeżeli argument składa się z mniejszej liczby znaków, to będzie ono uzupełnione do długości minimalnej odstępami. Jeżeli specyfikacja długości rozpoczyna się cyfrą `0`, to pole będzie wypełniane nie znaczącymi zerami;
- `.`⟨*ciąg-cyfr*⟩, maksymalna drukowana długość napisu lub liczba cyfr po kropce dziesiętnej.

Oto lista znaków przekształceń i ich znaczenie (zapis [-] oznacza, że znak `-` jest opcjonalny):

Znaki konwersji

Znak	Typ przekształcenia
`c`	znak
`d`	liczba całkowita
`e`	liczba postaci [-]`d.ddddddE`[-]`dd`
`f`	liczba postaci [-]`ddd.dddddd`
`g`	`e` lub `f`, w zależności od tego, które jest krótsze bez nieznaczących zer
`o`	liczba ósemkowa bez znaku
`s`	napis
`x`	liczba szestnastkowa bez znaku

Jeżeli znak występujący po `%` nie jest znakiem przekształcenia to jest on po prostu wypisany; zatem `%%` spowoduje wypisanie znaku `%`.

Poniższe zestawienie ilustruje działanie różnych specyfikacji. Aby można ocenić długości pól otoczono je znakami `|`, zaś spacje oznaczono znakiem `~`.

Przykłady specyfikacji

Specyfikacja	Argument	Wynik
`%5d%%`	`33.33`	`|~~~33%|`
`%c`	`33.33`	`|!|`
`%d`	`33.33`	`|33|`
`%5d`	`33.33`	`|~~~33|`
`%e`	`3.1415`	`|3.141500e+000|`
`%f`	`3.1415`	`|3.141500|`

`%8.3f`	`3.1415`	`|~~~3.141|`
`%08.3f`	`3.1415`	`|0003.141|`
`%s`	`Alibaba`	`|Alibaba|`
`%9s`	`Alibaba`	`|~~Alibaba|`
`%-9s`	`Alibaba`	`|Alibaba~~|`
`%-.3s`	`Alibaba`	`|Ali|`
`%-9.3s`	`Alibaba`	`|Ali~~~~~~|`

12.2. Instrukcja `print`

Instrukcja `print` jest uproszczoną formą `printf` a jej działanie jest następujące: drukowane argumenty są oddzielane wartością zmiennej wbudowanej `OFS` (standardowo `OFS=" "` – argumenty oddzielone są znakiem odstępu) a na końcu listy jest drukowana wartość zmiennej wbudowanej `ORS` (standardowo `ORS="\n"` – każde kolejne `print` drukuje od nowego wiersza). Wszystkie argumenty numeryczne są drukowane w oparciu o tę samą specyfikację, określoną poprzez wartość zmiennej `OFMT` (standardowo `OFMT="%.6g"`). Stąd, uruchamiając poniższy przykład:

```
awk 'BEGIN {OFS=":";ORS="->"; print log(2), log(3); print log(5); }'
```

otrzymamy:

```
0.693147:1.09861->1.60944->
```

UWAGI: `print` to skrót od `print $0` (a nie, jak można by się spodziewać, `print ORS`).

12.3. Drukowanie do plików

Zamiast na ekran (standardowe wyjście) instrukcje `printf`/`print` mogą przesłać dane do pliku. Służą do tego operatory `>` oraz `>>`, zaś instrukcje mają wówczas postać:

```
printf ⟨format⟩, ⟨arg1⟩,... > ⟨plik⟩
print ⟨arg1⟩,... > ⟨plik⟩
```

lub

```
printf ⟨format⟩, ⟨arg1⟩,... >> ⟨plik⟩
print ⟨arg1⟩,... >> ⟨plik⟩
```

⟨*plik*⟩ jest napisem lub zmienną typu napisowego zawierającą legalną (z punktu widzenia systemu operacyjnego) nazwę pliku. Przykładowo program:

```
{ printf "%s\n", $0 > $1 }
```

będzie działał doputy, dopóki wartością `$0` w pliku wejściowym będzie napis mogący być legalną nazwą pliku (oraz dopóki liczba otwartych jednocześnie plików nie przekroczy dopuszczalnego maksimum, porównaj uwagi z przykładu 22 (s. 41)).

UWAGI: Instrukcja `printf "%d %d\n", $1, $2 > $3` spowoduje wydrukowanie `$1` i `$2` do pliku określonego jako wartość pola `$3`, a nie `$1` i wyniku porówania wartości drugiego oraz trzeciego pola. Jeżeli chcemy osiągnąć to drugie to powinniśmy napisać: `printf "%d %d\n", $1, ($2 > $3)` albo `printf ("%d %d\n", $1, $2 > $3)`.

Operator `>` otwiera plik tylko raz (niszcząc poprzednią zawartość); kolejne instrukcje `printf` dodają tekst do tego pliku. Operator `>>` różni się od `>` tym, że przy otwarciu pliku poprzednia zawartość nie jest niszczona.

PRZYKŁAD 20

Przykład 12 (s. 29) zawiera program drukujący zestawienie obrotów dla win z podziałem według smaku i koloru. Poważną wadą programu jest niemożność określenia porządku, w jakim ma on drukować poszczególne pozycje zestawienia. Poniżej poprawiona wersja drukująca poszczególne rubryki w porządku alfabetycznym:

```
#!/bin/bash
if test $# -eq 0; then
  echo "‘basename $0‘: plik..." >&2; exit 0; fi

cat $* | awk ' {  o = $NF * $(NF-1);
   smaki[$(NF-2)] += o; kolory[$(NF-3)] += o;
   obrot[$(NF-2), $(NF-3)] += o
}
END { for (smak in smaki)
         for (kolor in kolory)
              if ((smak, kolor) in obrot)
                print smak, kolor, obrot[smak, kolor];
}' | sort | awk ' BEGIN { kreska = "+----------------------+"
     KRESKA = "+======================+"; print KRESKA; }
{ if (kolor != $1) {
    if (NR !=1) {
      printf "| Razem:   | %10.2f |\n", razem;
      print KRESKA; razem = 0}
      printf "| Wina %-16.16s |\n", $1;
      print kreska;
    }
  razem += $3; rrazem += $3;  kolor = $1;
  printf "| %-8.8s | %10.2f |\n", $2, $3
}
END { printf "| Razem:   | %10.2f |\n", razem;
      print KRESKA;
      printf "| OGÓŁEM:  | %10.2f |\n", rrazem;
      print KRESKA; } '
```

Problem został rozwiązany z wykorzystaniem *dwóch* skryptów AWK-owych oraz standardowego programu `sort`. Pierwszy skrypt dokonuje obliczeń a wyniki

drukuje w formacie odpowiednim dla programu `sort`, który jest uruchamiany w kolejnym kroku. Wydrukowaniem posortowanych danych w formie eleganckiej tabelki zajmuje się drugi skrypt AWK-owy. Całość została „zintegrowana” poprzez połączenie skryptów oraz programu `sort` w potok i umieszczenie wszystkiego w jednym skrypcie shellowym. (Skrypt ten oczekuje nazw plików do przeczytania jako argumentów wywołania. Instrukcja: `if test $# -eq 0;... fi` sprawdza czy podano jakieś argumenty; jeżeli nie to skrypt kończy działanie. Polecenie `cat $*` wysyła strumień danych na wejście pierwszego programu AWK-owego. Przypominamy, że `$*` w języku `bash` oznacza listę wszystkich argumentów, z którymi uruchomiono skrypt.) □

12.4. Drukowanie w potoku

Możliwe jest także drukowanie w potoku za pomocą instrukcji `printf` (`print`) o postaci:

`printf` ⟨*format*⟩`,` ⟨*arg1*⟩`,... |` ⟨*program*⟩
`print` ⟨*arg1*⟩`,... |` ⟨*program*⟩

⟨*program*⟩ jest napisem lub zmienną napisową zawierającą nazwę programu-odbiorcy strumienia danych drukowanych przez `printf` (`print`).

PRZYKŁAD 21
Jako ilustrację zmodyfikujemy dwuwierszowy program z przykładu 11 (s. 28), tak żeby poszczególne kategorie były drukowane według malejących obrotów:

```
{ obrot[$(NF-2)] += $NF * $(NF-1) }
END {for (wino in obrot) {print wino, obrot[wino] | "sort -nr +1" }}
```

Dla przypomnienia `sort -nr +1` oznacza sortowanie numeryczne, w odwrotnym porządku poczynając od drugiego pola. □

12.5. Funkcja `close`

Funkcja `close` zamyka plik otwarty za pomocą operatorów `>`, `>>` i `|`. Jej składnia jest następująca:

`close(`⟨*napis*⟩`)`

gdzie ⟨*napis*⟩ jest identyczny z napisem ⟨*plik*⟩ lub ⟨*program*⟩, za pomocą których uprzednio otwarto plik/potok. Funkcja ta jest niezbędna w sytuacji gdy np. najpierw piszemy do pliku a następnie chcemy (w obrębie tego samego programu) czytać z niego dane.

Napis zamykający plik lub potok musi być *identyczny z dokładnością do znaków odstępu*, z tym, za pomocą którego plik/potok został otworzony. Z tego względu zalecamy używanie do otwierania i zamykania plików i potoków uprzednio zadeklarowanych zmiennych a nie posługiwanie się stałymi napisowymi.

PRZYKŁAD 22

Należy przepisać zawartości pliku `tdf2000.txt` w taki sposób aby każdy etap został zapisany w odzielnym pliku. Poniższy program jest jednym z możliwych sposobów rozwiązania problemu:

```
#!/usr/bin/awk -f
NR==1, /^-+[ \t]*$/ { next }  # Pomiń nagłówek,
/^Etap[ \t]+[0-9]+:/ { close(File);
    sub(/:/,"",$2); File="etap" $2;
    next }
NF > 0 { print $0 > File}
```

Instrukcja `File="etap" sub(/:/,"",$2)` tworzy nazwę pliku według schematu `etap`⟨*numer-etapu*⟩. Kolejne niepuste wiersze (`NF > 0`) będą wysyłane do pliku `etap`⟨*numer-etapu*⟩. Polecenie `close(File)` zamyka plik związany z poprzednim etapem. W tym przykładzie `close` nie jest potrzebna bo będziemy pisać raptem do trzech plików. Gdyby jednak etapów było więcej to mogłoby się okazać, że próbujemy otworzyć więcej plików niż jest to dopuszczalne (dopuszczalna maksymalna liczba otwartych plików jest zależna od ustawień systemowych) i program zakończyłby się błędem.

Ważna jest kolejność: najpierw zamykamy plik (poprzedni) a dopiero w kolejnej iteracji otwieramy następny. Dla pierwszego etapu nie ma wprawdzie czego zamykać – ale w AWK próba zamknięcia pliku, który nie jest otwarty nie jest błędem. □

PRZYKŁAD 23

Aby obliczyć procentowy udział obrotu każdego wina w obrocie ogółem (plik `wina.txt`) należy przeczytać plik dwa razy. (Można też próbować wczytać cały plik do pamięci, ale takie rozwiązanie jest bardziej skomplikowane a także, dla bardzo długiej listy win, może zabraknąć pamięci.)

```
#!/bin/bash
if [ ! -f "$1" ]; then
echo "Wykorzystanie: `basename $0`: plik" >&2; exit 0; fi
awk 'FNR==1 { pass++ }
pass == 1 {oo += $NF * $(NF-1) }
pass == 2 {print $0, 100 * $NF * $(NF-1)/oo }' $1 $1
```

Trik polega na dwukrotnym umieszczeniu tego samego pliku jako argumentów wywołania AWK. Przypominamy, że zmienna `FNR` zawiera numer bieżącego rekordu, licząc w każdym pliku wejściowym od początku. Licznik `pass` jest zwiększany o 1 po napotkaniu pierwszego wiersza każdego pliku zatem ma on wartość 1 przy pierwszym i 2 przy drugim czytaniu pliku. W konsekwencji druga akcja będzie wykonywana dla każdego wiersza podczas jego pierwszego czytania zaś trzecia podczas drugiego. Takie rozwiązanie dwukrotnego czytania tego samego pliku pozwala znacznie uprościć program: nie musimy jawnie czytać pliku w pętli

`while` za pomocą `getline`, korzystać z funkcji `split` i zamykać plik za pomocą `close`. □

12.6. Funkcja `fflush`*

Funkcja `fflush`* o postaci:

`fflush(`⟨*napis*⟩`)`

opróżnia bufor związany z plikiem lub potokiem poprzez ⟨*napis*⟩, identyczny z napisem ⟨*plik*⟩ lub ⟨*program*⟩, za pomocą których uprzednio otwarto plik/potok. Dopuszczalne są także dwie następujące postacie funkcji: `fflush()` oraz `fflush("")`. Pierwsza opróżnia bufor związany ze standardowym wyjściem, druga bufory związane z *wszystkimi* otwartymi w danej chwili plikami/potokami.

12.7. Funkcja `system`

Funkcja `system` ma postać:

`system(`⟨*instrukcja*⟩`)`

wykonuje instrukcję systemową przekazaną jako wartość napisowego argumentu ⟨*instrukcja*⟩. Najczęściej funkcje `printf`/`printf` i operatory `>`, `>>` i `|` wystarczają do zrealizowania typowych zadań bez potrzeby uciekania się do funkcji `system`.

13. Argumenty wywołania programu

Wartości argumentów wywołania programu są, podobnie jak w wypadku języka C, przechowywane we wbudowanej zmiennej tablicowej `ARGV`. Zmienna `ARGC` zaś zawiera liczbę argumentów. Przykładowo:

```
gawk -v v=1 -f 1.awk test.txt v=1 b
```

`ARGC` ma wartość 4, `ARGV[0] ="gawk"`, `ARGV[1] ="test.txt"`, `ARGV[2] ="v=1"`, `ARGV[3] ="b"`. Opcje i ich wartości nie są liczone (w przykładzie są to `-f` i `-v`) a `ARGV[0]` jest równa nazwie uruchomionego programu (tu `gawk`). Poniższy program wypisuje wartość wszystkich argumentów wywołania:

```
BEGIN {for (i=0; i< ARGC; i++) {print ARGV[i]}  }
```

Zmienne `ARGC` i `ARGV` można zmieniać wewnątrz programu. Po napotkaniu końca bieżącego pliku wejściowego AWK pobiera następny element tablicy `ARGV` jako nazwę następnego pliku wejściowego. Przykładowo:

```
BEGIN{ ARGV[1]="test.txt"; if (ARGC < 2) {ARGC = 2} }
```

spowoduje, że AWK, *zawsze* będzie przeszukiwał jako pierwszy plik „`test.txt`” bez względu na to jaki (i czy w ogóle) podamy w linii poleceń.

Aby usunąć nazwę pliku z tablicy `ARGV` możemy albo nadać odpowiedniemu elementowi tablicy wartość napisu pustego, albo posłużyć się poleceniem `delete`. Napis „`-`” oznacza standardowe wejście. Tego typu manipulacje z reguły umieszczamy wewnątrz wzorca `BEGIN`, por. przykład 15 (s. 32).

PRZYKŁAD 24
W przykładzie 17 (s. 35) wielkość pliku była zadeklarowana na stałe co jest pewną niedogodnością; poniższa modyfikacja pozbawiona jest już tej wady:

```
BEGIN { flag = 1;
   if (ARGC > 1) {DUZY = ARGV[1] * 1000} else { DUZY = 0; };
   while ("ls -l" | getline > 0) {
     if (NF == 9  && $5 > DUZY) {print; flag =0 }}
   if (flag) {print "Nie ma plików większych od", DUZY; }
}
```

Chcąc uzyskać listę plików np. większych od 500kb, wystarczy teraz napisać `awk -f chkbig.awk 500` (gdzie `chkbig.awk` zawiera powyższy kod). □

14. Uruchamianie AWK

AWK można wywołać z kilkunastoma opcjami. Najważniejsze z nich to `-f`, `-F` oraz `-v`. Dwie pierwsze już omówiliśmy, ostatnia umożliwia nadanie zmiennej (ogólnie zmiennym, ponieważ możemy ją używać wielokrotnie) wartości w momencie uruchomienia programu (z linii poleceń). Przykład 25 (s. 43) ilustruje sposób wykorzystania opcji `-v`.

UWAGI: Z czasem gdy dorobimy się biblioteki własnych funkcji AWK-owych, może powstać problem, jak w elegancki sposób dołączać ją do różnych plików, w taki sposób, jak umożliwia to, np. instrukcja `#include` w C? Okazuje się, że opcja `-f` nie musi wcale występować tylko raz:

```
awk -f mylib.awk -f ⟨program⟩ ⟨plik⟩
```

gdzie plik `mylib.awk` zawiera bibliotekę naszych funkcji. Nie jest to rozwiązanie idealne ale – według nas – najlepsze z możliwych.

Zmienna środowiskowa AWKPATH zawiera katalogi, w których AWK szuka plików źródłowych podanych za pomocą opcji `-f`.

PRZYKŁAD 25
Problem z przykładu 17 (s. 35) można rozwiązać w inny sposób niż ten zaprezentowany w przykładzie 24 (s. 43) umieszczając zmodyfikowany program AWK-owy (oczywiste zmiany pozostawiamy czytelnikom) w skrypcie shellowym:

```
#!/bin/bash
awk -vDUZY=$1 'BEGIN { ⟨...⟩ }'
```

Żeby otrzymać listę plików większych od np. 600kb wystarczy teraz napisać: `chkbig 600` (gdzie `chkbig` jest nazwą skryptu). □

Bibliografia

[1] Aho Alfred V., Kernighan Brian W., Weinberger Peter J.: *The AWK Programming Language*, Addison-Wesley 1988.

[2] Aho Alfred V., Kernighan Brian W., Weinberger Peter J.: *AWK – A Pattern Scanning and Processing Language*, 1978. Dostępny m.in. w `http://www.softlab.ntua.gr/unix/docs/awk.ps` jako plik postscriptowy.

[3] *comp.lang.awk FAQ*. Dokument dostępny m.in. w `http://www.faqs.org/faqs/computer-lang/awk/faq`.

[4] Kernighan Brian W., Ritchie Denis M.: *Język C*, WNT 1988.

[5] Lichoński Bogusław, Przechlewski Tomasz: AWK – opis języka z przykładami, *Biuletyn GUST* 7/1996, s. 10–25.

[6] Robbins Arnold D.: *Effective AWK Programming. A User's Guide for GNU AWK, version 1.0.4*, April 1999. Podręcznik rozpowszechniany w pakiecie z programem `gawk`, dostępny m.in. w `ftp://sunrise.pg.gda.pl/pub/gnu/gawk`.

Skorowidz

Hasła oznaczone * oznaczają rozszerzenia standardu AWK, zaś oznaczone znakiem † funkcje zdefiniowane przez autora.

`#`, 10

Aho, Alfred, 5
`ARGC`, 42
`ARGIND`*, 16, 26
`ARGV`, 42
`atan2`, 20
AWKPATH, 43

`BEGIN`, 10, 43
`break`, 25
Brennan Michael, 6

`close`, 35, 40
`continue`, 25
`cos`, 20

`delete`, 28, 43
`do`, 25
DOS, 5, 22

`END`, 10, 26
`ENVIRON`*, 16
`ERRNO`*, 16
`exit`, 26
`exp`, 20

`fflush`*, 42
`FIELDWIDTHS`*, 16, 36
`FILENAME`, 16, 26
`FNR`, 16, 26, 34, 41
`for`, 25
`FS`, 7, 10, 16, 21, 31
funkcje, 29
— rekurencyjne, 30

`gensub`*, 16, 21
`getline`, 34, 35
`gsub`, 16, 20, 22

`if`, 25
`IGNORECASE`*, 16
`index`, 16, 21
`int`, 20
ISO-8859-1, 16

Kernighan, Brian, 5
komentarz, 10

`leapyear`†, 18
`length`, 21
`log`, 20

match, 16, 21
max†, 30
modulo, 17

next, 14, 26
nextfile*, 26
NF, 8, 16, 17

OFMT, 38
OFS, 17, 38
opcje wywołania, 43
ORS, 32, 38

print, 38
printf, 36

rand, 20
return, 30
RLENGTH, 21
Robbins, Arnold, 24, 30, 32
RS, 7, 16, 31–33
RSTART, 21
RT*, 32, 33

sin, 20
split, 16, 21
sprintf, 21
sqrt, 20
srand, 20
strftime*, 24
sub, 16, 17, 22
SUBSEP, 28
substr, 22
system, 42
systime*, 24

tablice
— asocjacyjne, 27, 29, 34
— wielowymiarowe, 28
tolower*, 22
toupper*, 22

upper†, 23

Weinberger, Peter, 5
while, 25
wyrażenie
— regularne, 12, 15
wzorzec, 5
— BEGIN/END, 9
— regularny, 11
— z przecinkiem, 10, 12
— złożony, 9, 11

zmienna
— globalna, 30
— lokalna, 30
— środowiskowa, 43
— wbudowana, 16